



有灵机器人

# C 与 C++ 语言编程规范与风格指南

整理：陆巍

Youling Robotics

2023 年 12 月 14 日

# 前言

本指南的内容来自多家企业、组织。其中规范是必须遵守的，而风格则可按照个人喜好选择，但同一项目中个人编写的代码应只使用一种风格。

# 目录

前言	i
第一章 注释	1
1.1 优秀的代码可以自我解释，不通过注释即可轻易读懂。	1
1.2 注释风格	1
1.3 一般规则	1
1.3.1 注释的内容要清楚、明了，含义准确，防止二义性。	1
1.3.2 注释解释代码难以直接表达的意图，而不是重复描述代码。	1
1.3.3 修改代码时，维护代码周边的所有注释，以保证注释与代码的一致性。不再有用的注释要删除。	3
1.3.4 注释应放在其代码上方相邻位置或右方，不可放在下面。如放于上方则需与其上面的代码用空行隔开，且与下方代码缩进相同。	3
1.3.5 对于 switch 语句下的 case 语句，如果因为特殊情况需要处理完一个 case 后进入下一个 case 处理，必须在该 case 语句处理完、下一个 case 语句前加上明确的注释。	3
1.3.6 避免在注释中使用缩写，除非是业界通用或子系统内标准化的缩写。	4
1.3.7 避免在一行代码或表达式的中间插入注释。	4
1.3.8 注释应考虑程序易读及外观排版的因素，使用的语言若是汉、英兼有的，建议多使用汉语，除非能用非常流利准确的英语表达。对于有外籍员工的，由产品确定注释语言。	4
1.3.9 标点、拼写和语法应按照所用语言的习惯进行书写。	4
1.3.10 注释格式采用 doxygen 可识别的格式。	4
1.3.11 TODO 待办注释	4
1.4 文件注释	4
1.5 类注释	5
1.6 函数注释	5
1.6.1 函数声明	6
1.6.2 函数定义	6

1.7	变量注释	6
1.7.1	类数据成员	7
1.7.2	全局变量（常量）	7
1.8	实现注释	7
1.8.1	代码前注释	7
1.8.2	行注释	7
1.8.3	函数参数注释	8
1.8.4	不要做的事	9
<b>第二章</b>	<b>头文件</b>	<b>10</b>
2.1	一些原则	10
2.1.1	头文件中适合放置接口的声明，不适合放置实现。	10
2.1.2	头文件应当职责单一	11
2.1.3	头文件应向稳定的方向包含	11
2.2	一些规则	12
2.2.1	每一个.c 文件应有一个同名.h 文件，用于声明需要对外公开的接口。	12
2.2.2	禁止头文件循环依赖。	13
2.2.3	.c/.h 文件禁止包含用不到的头文件。	13
2.2.4	头文件应当自包含	13
2.2.5	总是编写内部 #include 保护符（#define 保护）。	13
2.2.6	禁止在头文件中定义变量。	14
2.2.7	只能通过包含头文件的方式使用其他.c 提供的接口，禁止在.c 中通过 extern 的方式使用外部函数接口、变量。	14
2.2.8	禁止在 extern "C" 中包含头文件。	14
2.3	一些建议	16
2.3.1	一个模块通常包含多个.c 文件，建议放在同一个目录下，目录名即为模块名。为方便外部使用者，建议每一个模块提供一个.h，文件名为目录名。	16
2.3.2	如果一个模块包含多个子模块，则建议每一个子模块提供一个对外的.h，文件名为子模块名。	16
2.3.3	头文件不要使用非习惯用法的扩展名，如.inc。	16
2.3.4	同一产品统一包含头文件排列方式。	16
2.4	内联函数	17
2.5	-inl.h 文件	18
2.6	函数参数顺序	18

<b>第三章 命名</b>	<b>19</b>
3.1 通用命名规则 . . . . .	19
3.2 文件 . . . . .	19
3.3 类 . . . . .	19
3.4 变量 . . . . .	20
3.5 常量 . . . . .	20
3.6 方法/函数 . . . . .	21
3.7 命名空间 . . . . .	22
3.8 枚举 . . . . .	22
3.9 宏定义 . . . . .	22

# 第一章 注释

注释对于保持代码的可读性绝对重要。以下规则说明了应该注释的内容和位置。但请记住：虽然注释非常重要，但最好的代码是自文档化的。为类型和变量赋予合理的名称比使用晦涩难懂的名称要好得多，因为你必须通过注释来解释这些名称。

在写注释时，要为你的读者而写，即下一个需要理解你的代码的贡献者。然而现实中往往下一位读者就是你自己！

## 1.1 优秀的代码可以自我解释，不通过注释即可轻易读懂。

优秀的代码不写注释也可轻易读懂，注释无法把糟糕的代码变好，需要很多注释来解释的代码往往存在坏味道，需要重构。

这条规则所包含的内容实际上很多，这里只是简单的列出，细节则包含在编程中的方方面面。

## 1.2 注释风格

一律使用“`///`”（注意，是三条）来注释。即使内容跨越多行，也同样如此。可能大家在写大段注释时觉得这样不方便，是的，就是故意让人觉得不方便，以免把写注释当成写小说，注释要简洁。

## 1.3 一般规则

### 1.3.1 注释的内容要清楚、明了，含义准确，防止二义性。

### 1.3.2 注释解释代码难以直接表达的意图，而不是重复描述代码。

注释的目的是解释代码的设计意图、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。

对于实现代码中巧妙的、晦涩的、有趣的、重要的地方加以注释。

注释不是为了名词解释（what），而是说明用途（why）。

**示例**

如下注释纯属多余：

```
1 ++i; /// increment i
2 if (receive_flag) /// if receive_flag is TRUE
```

如下这种无价值的注释不应出现（空洞的笑话，无关紧要的注释）：

```
1 /// 时间有限，现在是:04，根本来不及想为什么，也没人能帮我说清楚
```

如下的注释则给出了有用的信息：

```
1 /// 由于 xx 编号网上问题，在 xx 情况下，芯片可能存在写错误，此芯片进行写操作后，必须
   ↪ 进行回读校
2 /// 验，如果回读不正确，需要再重复写-回读操作，最多重复三次，这样可以解决绝大多数网上
   ↪ 应用时
3 /// 的写错误问题
4 int time = 0;
5 do {
6     write_reg(some_addr, value);
7     time++;
8 } while ((read_reg(some_addr) != value) && (time < 3));
```

对于实现代码中巧妙的、晦涩的、有趣的、重要的地方加以注释，出彩的或复杂的代码块前要加注释：

```
1 /// Divide result by two, taking into account that x contains the carry from the
   ↪ add.
2 for (int i = 0; i < result->size(); i++)
3 {
4     x = (x << 8) + (*result)[i];
5     (*result)[i] = x >> 1;
6     x &= 1;
7 }
```

- 1.3.3 修改代码时，维护代码周边的所有注释，以保证注释与代码的一致性。不再有用的注释要删除。
- 1.3.4 注释应放在其代码上方相邻位置或右方，不可放在下面。如放于上方则需与其上面的代码用空行隔开，且与下方代码缩进相同。
- 1.3.5 对于 switch 语句下的 case 语句，如果因为特殊情况需要处理完一个 case 后进入下一个 case 处理，必须在该 case 语句处理完、下一个 case 语句前加上明确的注释。

这样比较清楚程序编写者的意图，有效防止无故遗漏 break 语句。

示例：

```
1  case CMD_FWD:
2      ProcessFwd();
3      /// now jump into case CMD_A
4  case CMD_A:
5      ProcessA();
6      break;
7      /// 对于中间无处理的连续 case，已能较清晰说明意图，不强制注释。
8  switch (cmd_flag) {
9      case CMD_A:
10     case CMD_B:
11         {
12             ProcessCMD();
13             break;
14         }
15         ...
16 }
```



1.3.6 避免在注释中使用缩写，除非是业界通用或子系统内标准化的缩写。

1.3.7 避免在一行代码或表达式的中间插入注释。

1.3.8 注释应考虑程序易读及外观排版的因素，使用的语言若是汉、英兼有的，建议多使用汉语，除非能用非常流利准确的英语表达。对于有外籍员工的，由产品确定注释语言。

1.3.9 标点、拼写和语法应按照所用语言的习惯进行书写。

1.3.10 注释格式采用 doxygen 可识别的格式。

1.3.11 TODO 待办注释

对那些临时的、短期的解决方案，或已经够好但并不完美的代码使用 TODO 注释。

这样的注释要使用全大写的字符串 TODO，后面括号（parentheses）里加上你的大名、邮件地址等，还可以加上冒号（colon）：目的是可以根据统一的 TODO 格式进行查找：

```
1  /// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
2  /// TODO(Zeke) change this to use relations.
```

如果加上是为了在“将来某一天做某事”，可以加上一个特定的时间（“Fix by November 2025”）或事件（“Remove this code when all clients can handle XML responses.”）。

## 1.4 文件注释

在每一个手工编写的文件开头加入文件注释。注释必须列出：版权说明、版本号、生成日期、作者姓名、内容、功能说明、修改日志等。格式如下：

```
1  /// -----
2  /// @file 本文件的文件名
3  /// @brief 本文件实现功能的简述
4  /// @details 详细描述
5  /// @copyright 版权声明
6  /// @author 作者
7  /// @version 版本
8  /// @date 创建日期
9  /// 修改日志
10 /// -----
```

示例：

```
1  /// -----
2  /// @file config.hpp
```

```
3  /// @brief configuration file
4  /// @copyright Copyright (c) 2008-2020, by Google. All rights reserved.
5  /// @author Charles Martin
6  /// @version 1.0.0
7  /// @date May 20, 2009
8  /// -----
```

对于开源项目，一般采用下面的方式来注释：

```
1  /// @copyright Copyright 2023 Willard Lu
2  ///
3  /// Use of this source code is governed by an MIT-style
4  /// license that can be found in the LICENSE file or at
5  /// https://opensource.org/licenses/MIT.
```

可以根据实际需要，添加各种说明，例如版本号。

通常，头文件要对所声明的类的功能和用法作简单说明，.cpp 文件包含了更多的实现细节或算法讨论，如果感觉这些实现细节或算法讨论对于阅读有帮助，可以把.cpp 中的注释放到.h 中，并在.cpp 中指出文档在.h 中。

不要单纯在.h 和.cpp 间复制注释，复制的注释偏离了实际意义。

## 1.5 类注释

每个非显而易见的类声明都应带有随附的注释，以说明其用途以及应如何使用它。格式如下：

```
1  /// @class 类名
2  /// @brief 类的简单概述
3  /// @ details 详细概述。
4  class Test {
5      ...
6  }
```

类注释应该为读者提供足够的信息，让他们知道如何以及何时使用类，以及正确使用类所必需的其他注意事项。如果类有任何同步前提需要说明。如果类的实例可以被多个线程访问，那么要特别注意记录多线程使用的规则和不变项。

如有需要，可在注释中附上一段简单的示例代码。

对于需要注释的枚举、结构，也使用同样的注释格式。

## 1.6 函数注释

函数声明处注释描述函数功能，定义处描述函数实现。

### 1.6.1 函数声明

几乎每个函数声明的前面都应该有注释，用来描述函数的功能和使用方法。只有当函数简单且明显时，这些注释才可以省略。注释使用描述式（"Opens the file"）而非指令式（"Open the file"）；注释只是为了描述函数而不是告诉函数做什么。通常，注释不会描述函数如何实现，那是定义部分的事情。

函数声明处注释的内容：

- 1) inputs（输入）及 outputs（输出）；
- 2) 如果函数分配了空间，需要由调用者释放；
- 3) 参数是否可以为 NULL；
- 4) 是否存在函数使用的性能隐忧（performance implications）；
- 5) 如果函数是可重入的（re-entrant），其同步前提（synchronization assumptions）是什么？

示例：

```
1  /// @brief 在光标的当前位置打印字符
2  ///
3  /// 如果字符是换行符（'\n'），光标应该移动到下一行（必要时滚动）。如果字符是一个回车
   ↩（'\r'），
4  /// 光标应该立即重置为当前行的开头，从而导致任何未来的输出覆盖该行上的任何现有输出。
5  /// 如果遇到退格符（'\b'），应该擦除前面的字符（在其上写一个空格并将光标移回一列）。
6  ///
7  /// @param ch 要打印的字符
8  /// @return 输入字符
9  int PutByte(char ch);
```

注意，不要过于啰嗦，也不要说完全显而易见的事情。

### 1.6.2 函数定义

如果某个函数的工作方式有些麻烦，则该函数定义应有一个解释性注释。例如，在定义注释中，可能会描述使用的任何编码技巧，概述所经历的步骤，或者解释为什么选择了以自己的方式实现功能而不是使用可行的替代方法。

注意，您不应该只是在.h 文件或其他地方重复函数声明中给出的注释。简单地概括一下函数的功能是可以的，但是注释的重点应该是它是如何实现的。

## 1.7 变量注释

通常变量名本身足以很好说明变量用途，特定情况下，需要额外注释说明。

### 1.7.1 类数据成员

每个类数据成员（也叫实例变量或成员变量）应注释说明用途，如果变量可以接受 NULL 或-1 等警戒值（sentinel values），须说明。

### 1.7.2 全局变量（常量）

全局变量要有较详细的注释，包括对其功能、取值范围以及存取时注意事项等的说明。例如：

```
1  /// The ErrorCode when SCCP translate
2  /// Global Title failure, as follows      变量作用、含义
3  /// 0 -SUCCESS      1 -GT Table error
4  /// 2 -GT error Others -no use           变量取值范围
5  /// only function SCCPTranslate() in
6  /// this modual can modify it, and other
7  /// module can visit it through call
8  /// the function GetGTTransErrorCode()  使用方法
9  BYTE g_GTTranErrorCode;
```

## 1.8 实现注释

对于实现代码中巧妙的、晦涩的、有趣的、重要的地方加以注释。

### 1.8.1 代码前注释

出彩的或复杂的代码块前要加注释，如：

```
1  /// Divide result by two, taking into account that x
2  /// contains the carry from the add.
3  for (int i = 0; i < result->size(); i++) {
4      x = (x << 8) + (*result)[i];
5      (*result)[i] = x >> 1;
6      x &= 1;
7  }
```

### 1.8.2 行注释

比较隐晦的地方要在行尾加入注释，可以在代码之后空两格加行尾注释，如：

```
1  /// If we have enough memory, mmap the data portion too.
2  mmap_budget = max<int64>(0, mmap_budget - index_->length());
```

```

3  if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock))
4  return; /// Error already logged.

```

注意，有两块注释描述这段代码，当函数返回时注释提及错误已经被记入日志。

前后相邻几行都有注释，可以适当调整使之可读性更好：

```

1  ...
2  DoSomething(); /// Comment here so the comments line up.
3  DoSomethingElseThatIsLonger(); /// Comment here so there are two spaces
4  /// between the code and the comment.
5  ...

```

### 1.8.3 函数参数注释

当函数参数的含义不明显时，请考虑以下补救措施之一：

- 如果参数是一个字面常量，并且在多个函数调用中以默认相同的方式使用相同的常量，那么应该使用一个命名常量来显式地表示约束，并确保它是有效的。
- 考虑更改函数签名，以用枚举参数替换布尔参数。这将使参数值能够自我描述。
- 对于具有多个配置选项的函数，可以考虑定义单个类或结构来保存所有选项，并传递一个实例。这种方法有几个优点。在调用点上，选项是通过名称引用的，这澄清了它们的含义。它还减少了函数的参数计数，这使得函数调用更容易读写。另一个好处是，当您添加另一个选项时，您不必更改调用点。
- 将大型或复杂的嵌套表达式替换为命名变量。
- 作为最后的选择，请使用注释来澄清调用点上的参数含义。

考虑下面的例子：

```

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

```

与：

```

1  ProductOptions options;
2  options.set_precision_decimals(7);
3  options.set_use_cache(ProductOptions::kDontUseCache);
4  const DecimalNumber product =
5      CalculateProduct(values, options, /*completion_callback=*/nullptr);

```

### 1.8.4 不要做的事

不要陈述显而易见的事情。特别是，不要逐字地描述代码的功能，除非此行为对精通 c++ 的读者来说并不明显。相反，应该提供更高级的注释来描述为什么代码要这样做，或者使代码自描述。

比较下面两段代码中的注释：

```
1  /// Find the element in the vector.  <-- Bad: obvious!
2  auto iter = std::find(v.begin(), v.end(), element);
3  if (iter != v.end()) {
4      Process(element);
5  }
```

```
1  /// Process "element" unless it was already processed.
2  auto iter = std::find(v.begin(), v.end(), element);
3  if (iter != v.end()) {
4      Process(element);
5  }
```

自描述代码不需要注释。来自上面例子的注释是显而易见的：

```
1  if (!IsAlreadyProcessed(element)) {
2      Process(element);
3  }
```

## 第二章 头文件

头文件的设计体现了大部分的系统设计。不合理的头文件布局是编译时间过长的根本原因，不合理的头文件实际上反映了不合理的设计。

**依赖：**这里特指编译依赖。若 `x.h` 包含了 `y.h`，则称作 `x` 依赖 `y`。依赖关系会进行传导，如 `x.h` 包含 `y.h`，而 `y.h` 又包含了 `z.h`，则 `x` 通过 `y` 依赖了 `z`。依赖将导致编译时间的上升。虽然依赖是不可避免的，也是必须的，但是不良的设计会导致整个系统的依赖关系无比复杂，使得任意一个文件修改都要重新编译整个系统，导致编译时间巨幅上升。

在一个设计良好的系统中，修改一个文件，只需要重新编译数个，甚至是一个文件。

某产品曾经做过一个实验，把所有函数的实现通过工具注释掉，其编译时间只减少了不到 10%，究其原因，在于 `A` 包含 `B`，`B` 包含 `C`，`C` 包含 `D`，最终几乎每一个源文件都包含了项目组所有的头文件，从而导致绝大部分编译时间都花在解析头文件上。

某产品更有一个“优秀实践”，用于将 `.c` 文件通过工具合并成一个比较大的 `.c` 文件，从而大幅度提高编译效率。其根本原因还是在于通过合并 `.c` 文件减少了头文件解析次数。但是，这样的“优秀实践”是对合理划分 `.c` 文件的一种破坏。

大部分产品修改一处代码，都得需要编译整个工程，对于 TDD 之类的实践，要求对于模块级别的编译时间控制在秒级，即使使用分布式编译也难以实现，最终仍然需要合理的划分头文件、以及头文件之间的包含关系，从根本上降低编译时间。

合理的头文件划分体现了系统设计的思想，但是从编程规范的角度看，仍然有一些通用的方法用来合理规划头文件。

### 2.1 一些原则

#### 2.1.1 头文件中适合放置接口的声明，不适合放置实现。

说明：头文件是模块（Module）或单元（Unit）的对外接口。头文件中应放置对外部的声明，如对外提供的函数声明、宏定义、类型定义等。

内部使用的函数（相当于类的私有方法）声明不应放在头文件中。

内部使用的宏、枚举、结构定义不应放入头文件中。

变量定义不应放在头文件中，应放在 `.c` 文件中。

变量的声明尽量不要放在头文件中，亦即尽量不要使用全局变量作为接口。变量是模块或单元的内部实现细节，不应通过在头文件中声明的方式直接暴露给外部，应通过函数接口的方式进行对

外暴露。即使必须使用全局变量，也只应当在.c 中定义全局变量，在.h 中仅声明变量为全局的。

### 2.1.2 头文件应当职责单一

说明：头文件过于复杂，依赖过于复杂是导致编译时间过长的主要原因。很多现有代码中头文件过大，职责过多，再加上循环依赖的问题，可能导致为了在.c 中使用一个宏，而包含十几个头文件。

示例：如下是某平台定义 WORD 类型的头文件：

```
1  #include <VXWORKS.H>
2  #include <KERNELLIB.H>
3  #include <SEMLIB.H>
4  #include <INTLIB.H>
5  #include <TASKLIB.H>
6  #include <MSGQLIB.H>
7  #include <STDARG.H>
8  #include <FIOLIB.H>
9  #include <STDIO.H>
10 #include <STDLIB.H>
11 #include <CTYPE.H>
12 #include <STRING.H>
13 #include <ERRNOLIB.H>
14 #include <TIMERS.H>
15 #include <MEMLIB.H>
16 #include <TIME.H>
17 #include <WDLIB.H>
18 #include <SYSLIB.H>
19 #include <TASKHOOKLIB.H>
20 #include <REBOOTLIB.H>
21 ...
22 typedef unsigned short WORD;
23 ...
```

这个头文件不但定义了基本数据类型 WORD，还包含了 stdio.h、syslib.h 等等不常用的头文件。如果工程中有 10000 个源文件，而其中 100 个源文件使用了 stdio.h 的 printf，由于上述头文件的职责过于庞大而 WORD 又是每一个文件必须包含的，从而导致 stdio.h/syslib.h 等可能被不必要的展开了 9900 次，大大增加了工程的编译时间。

### 2.1.3 头文件应向稳定的方向包含

说明：头文件的包含关系是一种依赖，一般来说，应当让不稳定的模块依赖稳定的模块，而当不稳定的模块发生变化时，不会影响（编译）稳定的模块。



就我们的产品（华为）来说，依赖的方向应该是：**产品依赖于平台，平台依赖于标准库**。某产品线平台的代码中已经包含了产品的头文件，导致平台无法单独编译、发布和测试，是一个非常糟糕的反例。

**除了不稳定的模块依赖于稳定的模块外，更好的方式是两个模块共同依赖于接口**，这样任何一个模块的内部实现更改都不需要重新编译另外一个模块。在这里，我们假设接口本身是最稳定的。

在敏捷开发中对此另有更透彻的说明，这里暂时参照华为与 Google 公司的内容。

## 2.2 一些规则

### 2.2.1 每一个.c 文件应有一个同名.h 文件，用于声明需要对外公开的接口。

说明：如果一个.c 文件不需要对外公布任何接口，则其就不应当存在，除非它是程序的入口，如 main 函数所在的文件。

现有某些产品中，习惯一个.c 文件对应两个头文件，一个用于存放对外公开的接口，一个用于存放内部需要用到的定义、声明等，以控制.c 文件的代码行数。不提倡这种风格。这种风格的根源在于源文件过大，应首先考虑拆分.c 文件，使之不至于太大。另外，一旦把私有定义、声明放到独立的头文件中，就无法从技术上避免别人 include 之，难以保证这些定义最后真的只是私有的。

本规则反过来并不一定成立。有些特别简单的头文件，如命令 ID 定义头文件，不需要有对应的.c 存在。

示例：对于如下场景，如在一个.c 中存在函数调用关系：

```
1 void foo() {  
2     bar();  
3 }  
4  
5 void bar() {  
6     Do something;  
7 }
```

必须在 foo 之前声明 bar，否则会导致编译错误。

这一类的函数声明，应当在.c 的头部声明，并声明为 static 的，如下：

```
1 static void bar();  
2  
3 void foo() {  
4     bar();  
5 }  
6  
7 void bar() {  
8     Do something;  
9 }
```

### 2.2.2 禁止头文件循环依赖。

说明：头文件循环依赖，指 a.h 包含 b.h，b.h 包含 c.h，c.h 包含 a.h 之类导致任何一个头文件修改，都导致所有包含了 a.h/b.h/c.h 的代码全部重新编译一遍。而如果是单向依赖，如 a.h 包含 b.h，b.h 包含 c.h，而 c.h 不包含任何头文件，则修改 a.h 不会导致包含了 b.h/c.h 的源代码重新编译。

### 2.2.3 .c/.h 文件禁止包含用不到的头文件。

说明：很多系统中头文件包含关系复杂，开发人员为了省事起见，可能不会去一一钻研，直接包含一切想到的头文件，甚至有些产品干脆发布了一个 god.h，其中包含了所有头文件，然后发布给各个项目组使用，这种只图一时省事的做法，导致整个系统的编译时间进一步恶化，并对后来人的维护造成了巨大的麻烦。

### 2.2.4 头文件应当自包含

说明：简单的说，自包含就是任意一个头文件均可独立编译。如果一个文件包含某个头文件，还要包含另外一个头文件才能工作的话，就会增加交流障碍，给这个头文件的用户增添不必要的负担。

示例：

如果 a.h 不是自包含的，需要包含 b.h 才能编译，会带来的危害：

每个使用 a.h 头文件的.c 文件，为了让引入的 a.h 的内容编译通过，都要包含额外的头文件 b.h。额外的头文件 b.h 必须在 a.h 之前进行包含，这在包含顺序上产生了依赖。

注意：该规则需要与“.c/.h 文件禁止包含用不到的头文件”规则一起使用，不能为了让 a.h 自包含，而在 a.h 中包含不必要的头文件。a.h 要刚刚可以自包含，不能在 a.h 中多包含任何满足自包含之外的其他头文件。

### 2.2.5 总是编写内部 #include 保护符（#define 保护）。

说明：多次包含一个头文件可以通过认真的设计来避免。如果不能做到这一点，就需要采取阻止头文件内容被包含多于一次的机制。

通常的手段是为每个文件配置一个宏，当头文件第一次被包含时就定义这个宏，并在头文件被再次包含时使用它以排除文件内容。

所有头文件都应该使用 #define 防止头文件被多次包含，命名格式为：

<PROJECT>\_<PATH>\_<FILE>\_H\_

为保证唯一性，头文件的命名应基于其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 按如下方式保护：

```
1  #ifndef FOO_BAR_BAZ_H_
2  #define FOO_BAR_BAZ_H_
```

```
3 ...  
4 #endif // FOO_BAR_BAZ_H_
```

注意：没有在宏最前面加上单下划线“\_”，是因为一般以单下划线“\_”和双下划线“\_\_”开头的标识符为 ANSI C 等使用，在有些静态检查工具中，若全局可见的标识符以“\_”开头会给出警告。

定义保护符时，应该遵守两条规则：一是保护符使用唯一名称；二是不要在受保护部分的前后放置代码或者注释。

例外情况：头文件的版权声明部分以及头文件的整体注释部分（如阐述此头文件的开发背景、使用注意事项等）可以放在保护符（`#ifndef XX_H_`）前面。

### 2.2.6 禁止在头文件中定义变量。

说明：在头文件中定义变量，将会由于头文件被其他.c 文件包含而导致变量重复定义。

### 2.2.7 只能通过包含头文件的方式使用其他.c 提供的接口，禁止在.c 中通过 `extern` 的方式使用外部函数接口、变量。

说明：若 a.c 使用了 b.c 定义的 `foo()` 函数，则应当在 b.h 中声明 `extern int foo(int input)`；并在 a.c 中通过 `#include <b.h>` 来使用 `foo`。禁止通过在 a.c 中直接写 `extern int foo(int input)`；来使用 `foo`，后面这种写法容易在 `foo` 改变时可能导致声明和定义不一致。

### 2.2.8 禁止在 `extern "C"` 中包含头文件。

说明：在 `extern "C"` 中包含头文件，会导致 `extern "C"` 嵌套，Visual Studio 对 `extern "C"` 嵌套层次有限制，嵌套层次太多会编译错误。在 `extern "C"` 中包含头文件，可能会导致被包含头文件的原有意图遭到破坏。例如，存在 a.h 和 b.h 两个头文件：

```
1 #ifndef A_H_  
2 #define A_H_  
3  
4 #ifdef __cplusplus  
5 void foo(int);  
6 #define a(value) foo(value)  
7 #else  
8 void a(int)  
9 #endif  
10  
11 #endif // A_H_
```

```
1 #ifndef B_H_  
2 #define B_H_  
3
```

```
4  #ifndef __cplusplus
5  extern "C" {
6  #endif
7
8  #include "a.h"
9  void b();
10
11 #ifndef __cplusplus
12 }
13 #endif
14
15 #endif // B_H_
```

使用 C++ 预处理器展开 b.h，将会得到

```
1  extern "C" {
2      void foo(int);
3      void b();
4  }
```

按照 a.h 作者的本意，函数 foo 是一个 C++ 自由函数，其链接规范为”C++”。但在 b.h 中，由于 #include ”a.h” 被放到了 extern ”C” 的内部，函数 foo 的链接规范被不正确地更改了。

示例：错误的使用方式：

```
1  extern "C"
2  {
3      #include "xxx.h"
4      ...
5  }
```

正确的使用方式：

```
1  #include "xxx.h"
2  extern "C"
3  {
4      ...
5  }
```

## 2.3 一些建议

### 2.3.1 一个模块通常包含多个.c 文件，建议放在同一个目录下，目录名即为模块名。 为方便外部使用者，建议每一个模块提供一个.h，文件名为目录名。

说明：需要注意的是，这个.h 并不是简单的包含所有内部的.h，它是为了模块使用者的方便，对外整体提供的模块接口。

以 Google test（简称 GTest）为例，GTest 作为一个整体对外提供 C++ 单元测试框架，其 1.5 版本的 gtest 工程下有 6 个源文件和 12 个头文件。但是它对外只提供一个 gtest.h，只要包含 gtest.h 即可使用 GTest 提供的所有对外提供的功能，使用者不必关心 GTest 内部各个文件的关系，即使以后 GTest 的内部实现改变了，比如把一个源文件 c 拆成两个源文件，使用者也不必关心，甚至如果对外功能不变，连重新编译都不需要。

对于有些模块，其内部功能相对松散，可能并不一定需要提供这个.h，而是直接提供各个子模块或者.c 的头文件。

比如产品普遍使用的 VOS，作为一个大模块，其内部有很多子模块，他们之间的关系相对比较松散，就不适合提供一个 vos.h。而 VOS 的子模块，如 Memory（仅作举例说明，与实际情况可能有所出入），其内部实现高度内聚，虽然其内部实现可能有多个.c 和.h，但是对外只需要提供一个 Memory.h 声明接口。

### 2.3.2 如果一个模块包含多个子模块，则建议每一个子模块提供一个对外的.h，文件名为子模块名。

说明：降低接口使用者的编写难度。

### 2.3.3 头文件不要使用非习惯用法的扩展名，如.inc。

说明：目前很多产品中使用了.inc 作为头文件扩展名，这不符合 c 语言的习惯用法。在使用.inc 作为头文件扩展名的产品，习惯上用于标识此头文件为私有头文件。但是从产品的实际代码来看，这一条并没有被遵守，一个.inc 文件被多个.c 包含比比皆是。本规范不提倡将私有定义单独放在头文件中，具体见规则 1。

除此之外，使用.inc 还导致 source insight、Visual studio 等 IDE 工具无法识别其为头文件，导致很多功能不可用，如“跳转到变量定义处”。虽然可以通过配置，强迫 IDE 识别.inc 为头文件，但是有些软件无法配置，如 Visual Assist 只能识别.h 而无法通过配置识别.inc。

### 2.3.4 同一产品统一包含头文件排列方式。

包含次序标准化可增强可读性、避免隐藏依赖（hidden dependencies），次序如下：C 库、C++ 库、其他库的.h、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，并且避免使用 UNIX 文件路径。（当前目录）和..（父目录）。例如，google-awesome-project/src/base/logging.h 应像这样被包含：

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

示例：dir/foo.cpp 的主要作用是执行或测试 dir2/foo2.h 的功能，foo.cpp 中包含头文件的次序如下：

dir2/foo2.h（优先位置，详情如下）

C系统文件

C++系统文件

其他库头文件

本项目内头文件

这种排序方式可有效减少隐藏依赖，我们希望每一个头文件独立编译。最简单的实现方式是将其作为第一个.h 文件包含在对应的.cpp 中。

dir/foo.cpp 和 dir2/foo2.h 通常位于相同目录下（像 base/basictypes\_unittest.cpp 和 base/basictypes.h），但也可在不同目录下。

相同目录下头文件按字母序是不错的选择。

举例来说，google-awesome-project/src/foo/internal/fooserver.cpp 的包含次序如下：

```
1 #include "foo/public/fooserver.h" // 优先位置
2
3 #include <sys/types.h>
4 #include <unistd.h>
5
6 #include <hash_map>
7 #include <vector>
8
9 #include "base/basictypes.h"
10 #include "base/commandlineflags.h"
11 #include "foo/public/bar.h"
```

## 2.4 内联函数

只有当函数只有 10 行甚至更少时才会将其定义为内联函数（inline function）。

**定义：**当函数被声明为内联函数之后，编译器可能会将其内联展开，无需按通常的函数调用机制调用内联函数。

**优点：**当函数体比较小的时候，内联该函数可以令目标代码更加高效。对于存取函数（accessor、mutator）以及其他一些比较短的关键执行函数。

**缺点：**滥用内联将导致程序变慢，内联有可能是目标代码量或增或减，这取决于被内联的函数的大小。内联较短小的存取函数通常会减少代码量，但内联一个很大的函数（译者注：如果编译器允

许的话)将戏剧性的增加代码量。在现代处理器上,由于更好的利用指令缓存(instruction cache),小巧的代码往往执行更快。

**结论:** 一个比较得当的处理规则是,不要内联超过 10 行的函数。对于析构函数应慎重对待,析构函数往往比其表面看起来要长,因为有一些隐式成员和基类析构函数(如果有的话)被调用!

另一有用的处理规则:内联那些包含循环或 switch 语句的函数是得不偿失的,除非在大多数情况下,这些循环或 switch 语句从不执行。

重要的是,虚函数和递归函数即使被声明为内联的也不一定是内联函数。通常,递归函数不应该被声明为内联的(译者注: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)。析构函数内联的主要原因是其定义在类的定义中,为了方便抑或是对其行为给出文档。

## 2.5 -inl.h 文件

复杂的内联函数的定义,应放在后缀名为-inl.h 的头文件中。

在头文件中给出内联函数的定义,可令编译器将其在调用处内联展开。然而,实现代码应完全放到.cpp 文件中,我们不希望.h 文件中出现太多实现代码,除非这样做在可读性和效率上有明显优势。

如果内联函数的定义比较短小、逻辑比较简单,其实现代码可以放在.h 文件中。例如,存取函数的实现理所当然都放在类定义中。出于实现和调用的方便,较复杂的内联函数也可以放到.h 文件中,如果你觉得这样会使头文件显得笨重,还可以将其分离到单独的-inl.h 中。这样即把实现和类定义分离开来,当需要时包含实现所在的-inl.h 即可。

-inl.h 文件还可用于函数模板的定义,从而使得模板定义可读性增强。

要提醒的一点是,-inl.h 和其他头文件一样,也需要 #define 保护。

## 2.6 函数参数顺序

定义函数时,参数顺序为:输入参数在前,输出参数在后。

C/C++ 函数参数分为输入参数和输出参数两种,有时输入参数也会输出(译者注:值被修改时)。输入参数一般传值或常数引用(const references),输出参数或输入/输出参数为非常数指针(non-const pointers)。对参数排序时,将所有输入参数置于输出参数之前。不要仅仅因为是新添加的参数,就将其置于最后,而应该依然置于输出参数之前。

这一点并不是必须遵循的规则,输入/输出两用参数(通常是类/结构体变量)混在其中,会使得规则难以遵循。

## 第三章 命名

### 3.1 通用命名规则

- 名称应该具有意义，能反映其作用，通常不要简化；
- 仅在缩写名称是公认、明确无歧义的情况下才使用，否则一律不使用缩写；
- 不得使用汉语拼音；
- 不要使用字母、数字、下划线之外的符号；
- 用正确的反义词组命名具有互斥意义的变量或相反动作的函数等；例如：  
add/remove begin/end create/destroy
- 尽量避免名字中出现数字编号，除非逻辑上的确需要编号；
- 标识符前不应添加模块、项目、产品、部门的名称作为前缀；
- 重构或修改部分代码时，应保持和原有代码的命名风格一致。

### 3.2 文件

- 文件名统一使用小写字母。这是因为不同操作系统对文件名大小写的处理上有区别，windows 系列忽略大小写，Linux 则区分大小写，所以干脆统一使用小写。
- 构成文件名的单词间使用下划线来连接。
- C++ 源文件使用.cpp 结尾，头文件使用.h 结尾。
- 定义类的文件一般成对出现，文件名称与类名称相对应，例如类 AnalyticalModel，与之对应的文件分别为：analytical\_model.h 与 analytical\_model.cpp。

### 3.3 类

每个单词以大写字母开头，不使用下划线。结构、typedef 与枚举也使用相同的命名规则。示例：

```
1 // classes and structs
2 class UrlTable { ...
3 class UrlTableTester { ...
```



```
4 struct UrlTableProperties { ...  
5  
6 // typedefs  
7 typedef hash_map<UrlTableProperties *, string> PropertiesMap;  
8  
9 // enums  
10 enum UrlTableErrors { ...
```

类的实例，也就是对象的命名与变量的命名规则相同。

### 3.4 变量

- 一律使用小写字母，单词间用下划线相连。类成员变量以下划线结尾。例如：

```
1 initial_position  
2 initial_position_
```

- 结构的数据成员与普通变量命名规则一样，但不用在结尾添加下划线。例如：

```
1 struct TimeType {  
2     IntType year;  
3     IntType month;  
4     IntType day;  
5     IntType hour;  
6     IntType minute;  
7     RealType second;  
8 };
```

- 全局变量使用“g\_”前缀，但通常情况下禁止使用全局变量。
- 静态变量使用“s\_”前缀。
- 禁止使用单个字母作为变量名，但允许定义 i、j、k 作为局部循环变量。
- 不得使用匈牙利命名法。因其有碍代码重构。
- 使用“名词”或“形容词 + 名词”的方式命名变量。

### 3.5 常量

字母全部大写，单词间用下划线连接。例如：

```
1 const int MINIMUM_NUMBER_OF_BYTES = 4;
```

应避免使用 `#define` 语句。应该使用 `const` 变量或枚举类型代替常量宏。（一个例外是当包含条件调试时可以使用 `#define` 语句）。

### 3.6 方法/函数

- 每个方法和函数都执行一个动作，因此名称应明确其作用。名称应为动词，并以大写开头的混合大小写形式。例如：

```
1 OutputCalibrationData()
2 Normalize()
```

- 推荐前缀：
  - Is/Has/Can：问一些问题并返回布尔类型
  - Set/Get
  - Initialize：初始化一个对象
  - Compute：计算一些东西（简单的），或者使用下面的
  - Calculate：计算一些东西（复杂的）
- 类的名称不应在方法名称中重复。例如：

```
1 Vector Normalize() // NOT: Vector NormalizeVector()
```

- 内联函数可以使用小写字母与下划线组合的方式命名。见下一条的示例。
- 存取函数（accessors and mutators）要与存取的变量名匹配。例如：

```
1 class MyClass {
2     public:
3         ...
4         int num_entries() const { return num_entries_; }
5         void set_num_entries(int num_entries) { num_entries_ = num_entries; }
6     private:
7         int num_entries_;
8 };
```

- C 函数使用小写字母与下划线组合的方式命名。例如：

```
1 get_best_fit_model()
2 load_best_estimate_model()
```

C++ 程序中使用 C 函数的情况很少，仅用于 C++ 和 C 代码之间的接口。

- 命名规则与变量一样。
- 传递类时，参数的名称可与其类型名称相同。但是，这不是必需的，在某些情况下甚至可能很麻烦。在这种情况下，名称应简洁。例如：

```
1 void SetForceModel(ForceModel *forceModel)
2 void SetForceModel(ForceModel *fm)
```

## 3.7 命名空间

命名空间的名称全小写，用下划线连接，其命名基于项目名称和目录结构。例如：`google__awesome__project`。

## 3.8 枚举

命名规则与类相同。枚举值应全部大写，单词间以下划线相连，例如：

```
1 enum Color {COLOR_RED, COLOR_GREEN, COLOR_BLUE};
```

## 3.9 宏定义

字母全部大写，用下划线连接，例如：

```
1 #define PI_ROUNDED 3.0
2 MY_EXCITING_ENUM_VALUE
```