编译原理实验编程规范（C 语言适用）

Use common sense and BE CONSISTENT.

1 介绍

本文档定义了一些编译原理课程实验中相关代码的规范，主要动机包括但不限于：

- 这是每一个从事代码工作的人的必备素质；
- 最大程度使他人能够尽快了解你的代码；
- 规范的代码风格能够让你专注比写代码更重要的事情；
- 与同行与时俱进，保持一致，便于交流；
- 便于自己、他人的维护、修改与更新，当然也方便我们的批改与检查；
- 在大型合作项目中需要统一的规则，减少合作者间的误会和混淆；
- 避免一些其他奇奇怪怪、危险的事情。

希望同学们能够尽量遵守，这很可能与实验的最终成绩有关。本文档完全参考了 Google C++ Style Guide。

2 头文件

通常每一个.c/.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.c 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。

2.1 自给自足

所有头文件要能够自给自足，用户和重构工具不需要为特别场合而包含额外的头文件。一个头文件要有 `#define` 保护，统统包含它所需要的其它头文件。

2.2 #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，命名格式当是:

`<PROJECT>_<PATH>_<FILE>_`

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

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_
…
#endif // FOO_BAR_BAZ_H_
```

2.3 内联函数

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。而滥用内联将导致程序变得更慢，内联一个相当大的函数将戏剧性的增加代码大小。一个较为合理的经验准则是，只有当函数只有 10 行甚至更少时才将其定义为内联函数。

2.4 #include 的路径及顺序

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

3 作用域

3.1 全局函数

尽量不要用裸的全局函数。

3.2 局部变量

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。

4 函数

4.1 函数参数

定义函数时，参数书写的顺序是先输入参数，再是输出参数（一些通过指针或引用类型传递的参数）。我们更倾向于输入参数为通产的值或是 const 的指针。

有时候，我们需要对函数的参数进行默认的设置（想一想 C++的中类构造函数），即保证没有函数能够以某种默认值（如 0）执行某种功能。

4.2 短小精悍

函数的功能要具体、完整且没有交叉，避免一个函数实现多种功能，建议一个函数做一件事情，如果还需要进行其他任务，交给一个新的小函数。

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：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，一致性更重要，所以不管你怎么想，规则总归是规则。

5.1 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。

推荐做法：

```
int price_count_reader;       // 无缩写
int num_errors;               // num 本来就很常见
int num_dns_connections;      // 人人都知道 DNS
```

不推荐做法：

```
int n;                        // 莫名其妙
int nerr;                     // 怪缩写
int n_comp_conns;             // 怪缩写
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;                 // 有删减若干字母。
```

5.2 文件命名

文件名要全部小写，可以包含下划线(_) 或连字符(-)。按项目约定来，如果并没有项目约定，下划线更好。

可接受的文件命名：

- `my_useful_function.c`
- `my-useful-function.c`
- `myusefulfunction.c`
- `muusefulfunction_test.c`

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.c`，对应于 FooBar 相关的代码。

内联函数必须放在 .h 文件中。如果内联函数比较短，就直接放在 .h 中。

5.3 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：`MyExcitingStruct`，`MyExcitingEnum`。

5.4 变量命名

变量名一律小写，单词之间用下划线连接，如：`a_local_variable`，`a_struct_data_member`。我们这里不推荐使用混合大小写的方式命名。

```
int student_number;  // 可 - 用下划线。
```

```
int studentnumber;   // 可 - 全小写。
int studentNumber;   // 不推荐 - 混合大小写。
```

5.5 常量命名

在全局的常量名称前加 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，无论是局部的或全局的，和其他变量稍作区别。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

常规函数的函数名的每个单词首字母大写, 没有下划线，使用大小写混合：

```
MyExcitingFunction(), MyExcitingMethod();
```

取值和设值函数则要求与变量名匹配（更多在 C++中出现）：

```
my_exciting_member_variable(), set_my_exciting_member_variable()
```

5.7 枚举命名

枚举的命名应当和常量或宏一致：kEnumName 或是 ENUM_NAME。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,
};
enum AlternateUrlTableErrors {
    OK = 0,
    OUT_OF_MEMORY = 1,
    MALFORMED_INPUT = 2,
};
```

5.8 宏命名

也许你并不会经常用到宏，但如果不得不用，其命名像枚举命名一样全部大写，使用下划线：

```
#define ROUND(x) ...
#define PI_ROUNDED 3.14
```

5.9 命名规则特例

如果将要命名的实体与已有 C/C++实体相似，可参考现有命名策略。

如使用 LONGLONG_MAX 来命名实体，表示一个与现有的 INT_MAX 同样性质的常量；用 bigopen()

这一函数名表示与 open()函数一致的函数。

6 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：下一个需要理解你的代码的人。慷慨些吧，下一个人可能就是你自己！

6.1 注释风格

使用 // 或者 /* */ 就好，在如何注释及注释风格上确保统一即可。通常习惯是，末尾的行注释// 与代码之间有两个空格。

6.2 文件注释

尽可能在每一个文件开头加入版权公告，然后是文件内容描述。版权公告主要包括法律公告和作者信息，如

版权声明：Copyright © 2017 Nanjing university

许可证：为项目选择合适的许可证版本，比如, Apache 2.0, BSD, LGPL, GPL

作者：标识文件的原始作者，包括 Email 等

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 文件要对所声明的结构体、函数的功能和用法作简单说明，.c/.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对于理解 .h 文件有帮助, 可以将该注释挪到 .h 中，并在 .c 中指出文档在 .h。注意，不要简单的在 .h 和 .c 文件间复制注释，这偏离了注释的实际意义。

6.3 函数注释

函数注释的准则：函数声明处注释描述函数功能，定义处描述函数实现。

6.3.1 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（“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主要有：函数的输入输出、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 等其他非法的值、是否存在函数使用上的性能隐患等。但也要避免啰嗦, 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();
```

6.3.2 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。注意不要从 .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的具体细节上。

6.4 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。结构体中变量的含义需要指明，对于其中的某些指针结构要指出是否可为 NULL 等值。对于全局变量，也要注释说明含义及用途。

6.5 实现注释

对于代码中巧妙的、晦涩的、有趣的、重要的地方加以注释。请注意，尽量不要使用自然语言对代码进行无意义的翻译。

6.5.1 代码前注释

巧妙或复杂的代码段前要加注释：

```
// 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;
}
```

6.5.2 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：

```
// 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.
```

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();  // Comment here so there are 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.
}
```

6.5.3 函数参数注释

向函数传入 NULL，布尔值或常量时，要注释说明含义，或通过常量命名规则让代码望文知意。

```
bool success = CalculateSomething(interesting_value,
                                  10,    // Default base value.
                                  false,  // Not the first time we're calling this.
                                  NULL);  // No callback.
```

如果你的函数参数具有比较复杂的结构，如使用了结构体的某个成员，最好定义一个新的变量来指明其含义。

6.6 标点，拼写和语法

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。

正确的标点，拼写和语法对此会有所帮助。通常，这些工作由一个好的 IDE 负责。

6.7 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大名，邮件地址，或其它身份标识,这里的冒号是可选的。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。

```
// TODO(genius@gmail.com): Use a "*" here for concatenation operator.
// TODO(your name) 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 April 2017”，或者一个明确的事项：“Remove this code when all clients can handle XML responses.”。

6.8 弃用注释

通过弃用注释（DEPRECATED comments）以标记某接口点或某些代码段已弃用。记得在 DEPRECATED 一词后，留下名字，邮箱地址以及其他补充内容。

6.9 注释技巧

1. 关于注释风格，很多 C++的 coders 更喜欢行注释，C coders 或许对块注释依然情有独钟；

2. 文件注释可以炫耀你的成就，也是为了捅了篓子别人可以找你；
3. 注释要言简意赅，不要拖沓冗余，复杂的东西简单化和简单的东西复杂化都是要被鄙视的；
4. 对于中国程序员来说，用英文注释还是用中文注释，是一个问题。但不管怎样，注释是为了让别人看懂，难道是为了炫耀编程语言之外的你的母语或外语水平吗？
5. 注释不要太乱，适当的缩进才会让人乐意看。但也没有必要规定注释从第几列开始 UNIX/LINUX 下还可以约定是使用 tab 还是 space，这可以通过编辑器设置自动实现；
6. TODO 很不错，有时候，注释确实是为了标记一些未完成的或完成的不尽如人意的地方，这样一搜索，就知道还有哪些活要干。

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

理论上说，过去遵循每一行代码字符数不超过 80 的规则，那是因为显示器没有那么宽，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，并且有了 80 列宽的传统标准。

7.2 编码

即使是英文，非 ASCII 字符也要尽力避免。特殊情况下可以适当包含此类字符，在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。此外，请尽可能使用 UTF-8 编码，因为很多工具都可以正确理解和处理 UTF-8 编码。

7.3 空格还是制表位

比较推荐 2 个空格的做法。通常制表位的宽度与四个空格相当，当你的代码层次很深，嵌套很多时，可能会使某些代码的缩进太多，代码左侧留白过多看起来费劲。通过 IDE 或其他文本编辑器设置缩进 tab 为 2 个空格的宽度，一劳永逸。

7.4 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。看上去像这样：

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

有时候甚至连第一个参数都放不下：

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

还有一些其他的情况：

1. 如果返回类型和函数名在一行放不下，分行，如果这么做了，不要缩进；
2. 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，左大括号总在最后一个参数同一行的末尾处；
3. 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，空函数体的右大括号和左大括号间总是有一个空格；
4. 函数声明和定义中的所有形参必须有命名且一致；
5. 所有形参应尽可能对齐，缺省缩进为两个空格，换行后的参数保持四个空格缩进。

7.5 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，后面每一行都和第一个实参对齐，左圆括号后和右圆括号前不要留空格：

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

参数也可以放在次行，缩进四格：

```
if (...) {
  ...
```

```
  ...
  if (...) {
    DoSomething(
        argument1, argument2,  // 4 空格缩进
        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);
```

或者放着不管，补充上注释：

```
bool retval = DoSomething(scores[x] * y + bases[x],  // Score heuristic.
                          x, y, z);
```

如果某参数独立成行，对可读性更有帮助的话，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，这里的参数可以表示为一个 3x3 的矩阵形式：

```
Transform(x1, x2, x3,
          y1, y2, y3,
          z1, z2, z3);
```

7.6 条件语句

对基本条件语句有两种可以接受的格式：一种在圆括号和条件之间有空格，另一种没有。最常见的是没有空格的格式，哪种都可以，但保持一致性，我们建议没有空格的形式。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。如果你还在徘徊的话，就不要加空格了。这是较为推荐的风格：

```
if (condition) {  // 圆括号里没空格紧邻。
  ...  // 2 空格缩进。
} else {  // else 的左括号与 if 的右括号同一行。
  ...
}
```

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。当然，也有一些项目要求 if 必须总是使用大括号。如果能增强可读性，简短的条件语句允许写在同一行，只当语句简单并且没有使用 else 子句时使用：

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

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，保持一致性。

7.7 循环语句

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 - continue 表明没有逻辑。
```

7.8 Switch 语句

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，请遵照如下格式：

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

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。

7.9 指针和引用表达式

指针或地址操作符 (*, &) 之后不能有空格，访问成员的句点或箭头前后不要有空格，这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;
```

而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，我们推荐与变量名放在一起。对于 int* a, b，新手可能会误认为 b 也是 int* 变量，但写成 int *a, b 就不一样了，高下立判。

```
// OK，空格前置。
char *c;
const string &str;

// OK，空格后置。
char* c;    // 但别忘了 "char* c, *d, *e, ...;"!
const string& str;
```

7.10 布尔表达式

如果一个布尔表达式超过标准行宽，断行方式要统一。下例中，逻辑与&&操作符总位于行尾，这在 Google 里很常见：

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

把所有操作符放在开头也可以。可以考虑额外插入圆括号，合理使用的话对增强可读性是很有帮助的。此外直接用符号形式的操作符，比如 && 和 ||，不要用词语形式的 and 和 or。

7.11 函数返回值

return 表达式里时没必要都用圆括号。如果你的返回值表达式较为复杂，也可以加上括号，增强可读性。

```
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);                 // 不推荐，毕竟你应该不会写 var = (value);
return(result);                 // 不推荐，return 可不是函数!
```

7.12 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。

```
// directives at beginning of line
  if (lopsided_score) {
#if DISASTER_PENDING      // 正确，从行开头起。
    DropEverything();
#endif
    BackToNormal();
  }
```

7.13 留白

7.13.1 水平留白

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。常规做法：

```
void f(bool b) {  // 左大括号前恒有空格。
  ...
int i = 0;  // 分号前不加空格。
int x[] = { 0 };  // 大括号内部可与空格紧邻也不可，不过要么两边都要加上。
int x[] = {0};  // 要么两边都没有
```

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

循环和条件语句：

```
if (b) {          // if 条件语句和循环语句关键字后均有空格。
} else {          // else 前后有空格。
}
while (test) {}   // 圆括号内部不紧邻空格。
switch (i) {
```

```
for (int i = 0; i < 5; ++i) {
switch ( i ) {    // 循环和条件语句的圆括号里可以与空格紧邻。
if ( test ) {     // 圆括号，但这很少见。总之要一致。
for ( int i = 0; i < 5; ++i ) {
for ( ; i < 5 ; ++i) {  // 循环里内 ; 后恒有空格，; 前可以加个空格。
switch (i) {
  case 1:         // switch case 的冒号前无空格。
    ...
  case 2: break;  // 如果冒号有代码，加个空格。
```

操作符：

```
// 赋值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，不过对 factors 前后不加空格也可以。
// 圆括号内部不紧邻空格。
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)
  ...
```

7.13.2 垂直留白

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，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。

总之，垂直留白越少越好。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。

7.14 其他

1. 关于 UNIX/Linux 风格为什么要把左大括号置于行尾（.c 文件的函数实现处，左大括号位于行首），这样代码看上去比较简约，想想行首除了函数体被一对大括号封在一起之外，只有右大括号的代码看上去确实也舒服；Windows 风格将左大括号置于行首的优点是匹配情况一目了然。我们倡导保持一致即可。

2. Google 强调有一对 `if-else` 时，不论有没有嵌套，都要有大括号。Apple 正好有栽过跟头。
3. 事实上，如果您熟悉英语本身的书写规则，就会发现这里的风格指南在格式上的规定与英语语法相当一脉相承。比如普通标点符号和单词后面还有文本的话，总会留一个空格；特殊符号与单词之间就不用留了，比如 `if (true)` 中的圆括号与 `true`。

8 结束语

这并不是一个强制的规定，但是如果你能够遵照这份规范，不仅提高你个人的编程效率，也方便他人的阅读、修改和判定。也许这一份文档看起来很长，事实上大部分的规定你在日常工作、学习中已经照做了，所以并不需要花费过多额外的时间，这样大家也可以把精力集中在实现内容而不是表现形式上。总之，Use common sense and *BE CONSISTENT*.

好了，关于代码风格的东西写的已经够多了，代码本身才更有趣，尽情享受吧！也希望各位同学能够享受编译原理的实验过程。