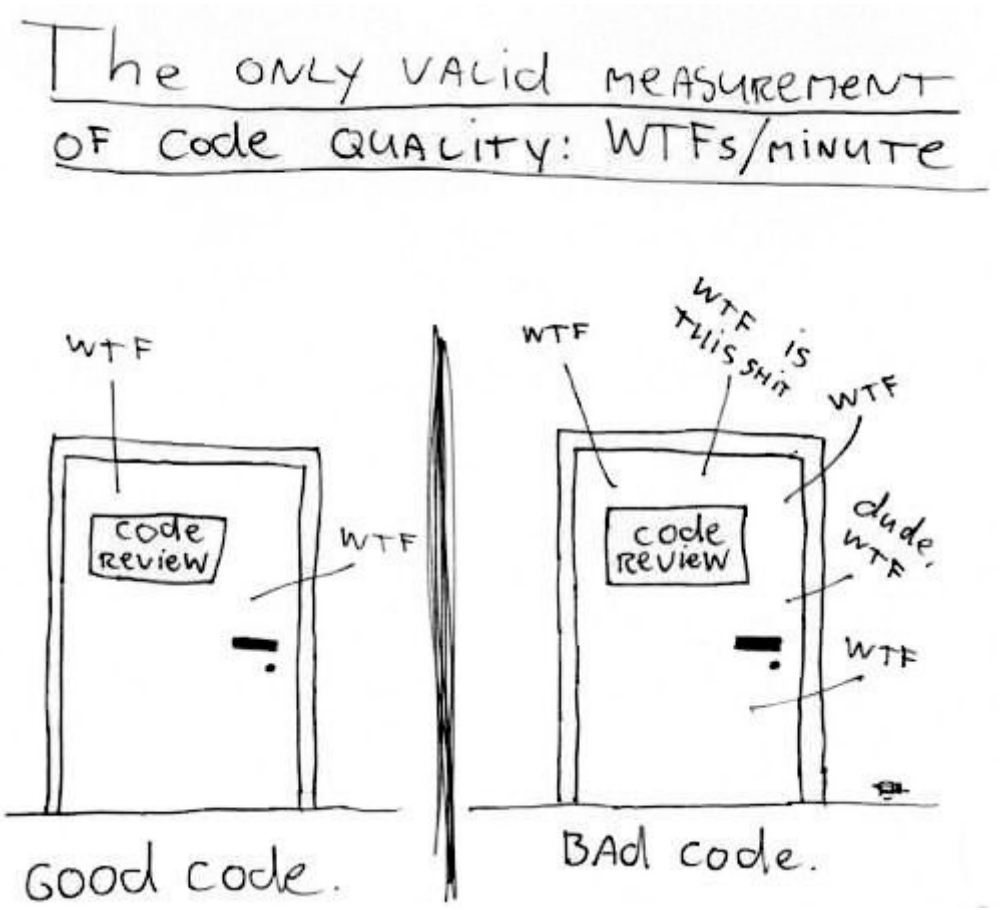
谈谈C语言的编程规范

C语言的“编程规范”在那些教材中很少提及，但值得本文花大篇幅特别补充。什么是“编程规范”呢？一图以蔽之：



每个人写代码都有独特的习惯，包括大括号位于行尾党和换行党（即所谓K&R风格和Allman风格），缩进的4空格、2空格和Tab党，更深一步还有更复杂的程序设计流派，彼此争论不休。想象一下两个代码风格迥然不同的程序员共同开发一个项目，会写出来什么样的代码？为了防止这种惨剧的发生，团队开发前一般都会为团队制定统一的代码规范。

一个好的程序编写规范是编写高质量程序的保证。清晰、规范的程序不仅仅是方便阅读，更重要的是能够写出更不容易出错（less error-prone）的代码，以及便于你或者共同开发的同事来检查错误、提高调试效率，从而最终**保证程序的质量和可维护性。**

> 整洁的代码如同优美的散文。—— Grady Booch
>
> 任何一个傻瓜都能写出计算机可以理解的代码。唯有写出人类容易理解的代码，才是优秀的程序员。—— Martin Fowler
>
> A standard says what will work, and how. It does not say what constitudes good and effective use. There are significant differences between understanding the technical details of programming language features and using them effectively in combination with other features, libraries, and tools to produces better software. By "better" I mean "**more maintainable, less error-prone, and faster**". We need to develop, popularize, and support conherent programming styles. Futher, we must support the evolution of older code to these more modern, effective, and coherent styles. —— Bjanre Stroustrup

首先要明确，在同一个编程语言中，代码规范也有不同标准；不同的团队，代码规范也大可不必相同。但所有的编程规范的目标都是为了程序的清晰性、简洁性。

例如如下这些：

- 华为的C语言编程规范：华为C语言编程规范 | wiki (gitbooks.io)
- Linux的编程规范：https://www.kernel.org/doc/html/v4.10/process/coding-style.html
- Nginx的编程规范：Development guide (nginx.org)

反面典型如：trekhleb/state-of-the-art-shitcode: 🤷‍♂️ State-of-the-art shitcode principles your project should follow to call it a proper shitcode (github.com)

在此，我简单列出部分良好的代码规范（或者说代码风格）：

变量命名

两个原则：名字与意义匹配，且易于区别。

我们不推荐a, b, c, x, y, z, ll, haizi,Fred,asdf,jkj这样的不明意义的命名，这些东西是破坏代码可读性的一大元凶。要尽量描述清楚对象的意义，能让人一眼看懂这是什么东西。

```
int a = 42; // Bad!
int age = 42; // Good!
```

另外，一般来说推荐的命名方法有下划线法和大驼峰法。如下：

```
const char* n = "Goldfish"; // Bad!

const char* my_first_name = "Goldfish"; // 下划线命名法
const char* MyFirstName = "Goldfish"; // 大驼峰命名法（又称帕斯卡命名法）
const char* myFirstName = "Goldfish"; // 小驼峰命名法
```

命名是还要注意要避免过于相似，避免"1" "l" "L"，"O" "o" "0"这种易混淆的情况，不要出现仅靠大小写区分的相似的标识符。更要避免重名，即使是不同级别的作用域。在一个程序中，命名方式要统一。

宏的名字只用大写字母（这不是必须的，但是这个约定俗成的规则已经被C程序员们遵守几十年了）。

注释

```
const int cdr = 700; // Bad!

// Good:

// The number of 700ms has been calculated empirically based on UX A/B test
results.
// @see: <link to experiment or to related JIRA task or to something that
explains number 700 in details>
const int callbackDebounceRate = 700;
```

注释是个好习惯。/* 下面的建议主要用于大型项目，而非日常作业的程序 */

- 文件头注释
- - 作者，文件名称，文件说明，日期，版权，许可证……
- 函数注释
- - 关键函数写上注释，说明函数的用途。
  - 特别函数参数，需要说明参数含义等。
  - 除了特别情况，注释写在代码之前，不要放到代码行之后。

- 关键代码或需要引起注意的代码，加上注释。
- 对于较大的代码块结尾，如for，while等，可加上 // end for或者end while
- ……
- `doxygen-enabled`注释（自行去了解）

注意：注释应该是说明“why”而不是“what”：

```
// Bad:
int error = 0; // Declares an integer called 'error'
// Good:
int error = 0; // Indicates whether an error occurred in XXX operation.
```

合适的空格和缩进

反面典型是某些ACM的风格（无冒犯）：(11 条消息) 请问我这样的代码风格丑吗？ - 知乎 (zhihu.com)

随便从洛谷上找一个（出处：记录详情 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)）

```
// Not Really Good:
#include<stdio.h>
#include<math.h>
int gcd(int a, int b) {
    return b ? gcd(b, a % b) : a;
}
int main() {
    int x, y, s = 0;
    scanf("%d%d", &x, &y);
    for (int i = x; i <= sqrt(x*y); i +=x)
        if(gcd(x*y/i,i)==x&&x*y%i==0) s+=2;
    if (x == y) s=1;
    printf("%d", s);
    return 0;
}

// Methinks much better :
#include <stdio.h>
#include <math.h>

/* Using Euclid's algorithm to calculate the maximum common factor. */
int gcd(int a, int b)
{
    int r = a % b;
    while (r)
    {
        a = b;
        b = r;
        r = a % b;
    }
    return b;
}
int main()
{
    int x, y;
    int ans = 0;
```

```
    scanf("%d %d", &x, &y);
    if (x == y)
    {
        ans = 1;
    } else {
        const int upper_bound = (int)sqrt(x * y);
        for (int factor = x; factor <= upper_bound; factor += x)
        {
            if (x * y % factor == 0 && gcd(factor, x * y / factor) == x)
            {
                /* Factor found. */
                ans += 2; //(factor1, factor2) and (factor2, factor1) both
counts.
            }
        }
    }
    printf("%d\n", ans);
    return 0;
}
```

想看更加Obfuscated的C语言代码，请关注IOCCC。

其他杂项

循环：

1. if 、for、while的循环即即使只有一句，也要加大括号（更能避免出错，便于日后可能的修改）。
2. 对于死循环，很多人常用while (true) 或者 while (1)；不过有一种观点认为这种死循环额外引入了一个变量（true或者1），他们更喜欢用for (; ;)来写死循环，并认为这在语义上更适合。
3. 循环过程尽量不要动循环计数器。
4. 避免“三角形”程序，即太多嵌套：

```
// Bad!
void someFunction() {
  if (condition1) {
    if (condition2) {
        if (result) {
          for (;;) {
            if (condition3) {
            }
          }
        }
    }
  }
}
// Not so bad:
void someFunction() {
  if (!condition1 || !condition2) {
    return;
  }

  if (result) {
    return;
  }

  for (;;) {
    if (condition3) {
```

```
    }
  }
  return;
}
```

函数：

1. 尽量短小且职责单一
2. 函数声明定义调用中，如果变量名过多，一行放不下，则分行放。可以与第一个变量对齐，或者空四格。、
3. 在一个（较长的）函数体内，逻辑上密切相关的语句之间不加空行，其它地方应加空行分隔。
4. 除了void函数，构造函数，析构函数，其它函数都要有返回值。（在一些编译器中，main函数不写return可能是允许的，但不建议）。
5. 注意：将数组作为参数进行传参时，数组会退化成指针。
6. 单一出口原则（非必须，但对于一个很长的函数，还是建议这么做的）。
7. 多次重复代码抽象成一个函数。即

代码块：

1. 一行代码只做一件事情，如只定义一个变量，或只写一条语句。这样的代码容易阅读，并且方便于写注释。
2. 如果你在同一个表达式中两次读取同一个变量，并且还对该变量进行写操作，那么结果很可能是未定义（Undefined）的。不要这么干。

   ```
   i++ + i++ ;  // Very Bad!
   f(v[i],i++); // Very Bad!
   v[i] = i++;  // Very Bad!
   ```

3. 不建议一次定义多个变量。

   ```
   int *p, q, r, *s, v[100]; // Bad!
   ```

4. 一行代码的最大长度宜控制在一定个字符以内，能在当前屏幕内全部可见为宜。
5. 定义变量完成后建议立即初始化。（RAII原则）

switch语句：

1. case子语句如果有变量，应用{}包含起来。(同时尽量避免这么做)
2. 为所有switch语句提供default分支。
3. 通常在case处理完毕之后需要按照程序设定的逻辑退出switch块，即必须添加break语句。如果你希望在某个case后不写break，那么要注释说明（即：you really mean it）。

类型：

1. 尽量不要出现类型转换！如必须，请显式写明类型转换！（一是类型转换不安全，二是编译器的strict-alias规则）
2. 注意float不一定比double节省空间，short不一定比int省空间。
3. 尽量避免用 union、volatile、goto……除非你知道自己在做什么。
4. （非必须，见仁见智）减少对宏的使用，因为宏只是暴力的替换，没有类型检查，编译器也不会看到你的宏定义，编译器的报错可能让你想不到是宏出现了问题。见：https://www.stroustrup.com/bs_faq2.html#macro。可以用 enum或者const变量进行替换。
5. （主要用于大型项目）为确保兼容性，请使用stdint.h中的类型。

杂项：

- 避免生成临时对象，尤其是大的临时对象。避免对象拷贝，尤其是代价很高的对象拷贝。

- 尽量在for循环之前，先写计算估值表达式。

```
for(int i = 0; i < strlen(s); ++i) { ... } // Bad! 每一次循环都会调用一次strlen

int len = (int)strlen(s); // strlen返回类型是size_t哦，这里有个小小的类型转换
for(int i = 0; i < len; ++i) { ... } // Good!
```

- 注意运算溢出问题。
- 不要假设表达式的运算顺序、函数参数的计算顺序等。
- 嵌套do-while(0)宏：目的是将一组语句变成一个语句，避免被其他if等中断。
- （非必须，见仁见智）尽量少使用静态变量（静态变量的生存周期和作用域不匹配，个人很不喜欢）。同理，不要滥用全局变量。
- （大型合作项目中）始终尊重项目或库中已经使用的代码风格
-

相关阅读：

- RobertC.Martin， Clean Code（罗伯特·马丁，《代码整洁之道》）
- Martin Fowler , Refactoring: Improving the Design of Existing Code（福勒《重构：改善既有代码的设计》）
- Beautiful Code: Leading Programmers Explain How They Think （《代码之美》）
- 前文提到的Linux的编程规范：https://www.kernel.org/doc/html/v4.10/process/coding-style.html
- John Ousterhout， A Philosophy of Software Design
- 6 ways minimalism can help you write clean code - DEV Community
- Google C++ Style Guide