火龙果 · 整理
uml.org.cn

嵌入式系统中的 C 编程规范

该文档规定了嵌入式系统中的 C 语言编程规范，包括了该文档的目的和意义；C 编程语言中的文件结构、命名规则、程序板式、表达式和基本语句、常量、函数设计等内容。但由于当前的编程环境大多是 C++/C 兼容的，所以也部分涉猎了 C++的内容。

火龙果 · 整理
uml.org.cn

目 录

1. 引言.....1
1.1. 嵌入式系统的定义.....1
1.2. 本文档的目的.....1
2.文件结构.....2
2.1. 版权和版本的声明.....3
2.2. 头文件的结构.....3
2.3 定义文件的结构.....4
2.4. 头文件的作用.....5
2.5.从编译器的原理出发理解.C 与.H 文件.....5
2.6 目录结构.....7
3. 命名规则.....8
3.1 共性规则.....8
3.2 扩展规则.....9
3.3 嵌入式系统的数据类型命名.....11
4. 程序的版式.....12
4.1. 空行.....12
4.2. 代码行.....13
4.3. 代码行内的空格.....14
4.4. 对齐.....15
4.5. 长行拆分.....16
4.6. 修饰符的位置.....17
4.7. 注释.....17
4.8. 使用的端口说明.....18
5. 表达式和基本语句.....19
5.1. 运算符的优先级.....19
5.2. 复合表达式.....20
5.3. if 语句.....21
5.3.1. 布尔变量与零值比较.....21
5.3.2. 整型变量与零值比较.....21
5.3.3. 浮点变量与零值比较.....22
5.3.4. 指针变量与零值比较.....22

5.3.5. 对 if 语句的补充说明 22
5.4. 循环语句的效率 23
5.5. for 语句的循环控制变量 24
5.6. switch 语句 24
5.7. goto 语句 25
6.常量 26
6.1. 为什么需要常量 26
6.2. const 与 #define 的比较 26
6.3 常量定义规则 26
7. 函数设计 27
7.1. 参数的规则 27
7.2.返回值的规则 28
7.3. 函数内部实现的规则 31
7.4 其它建议 32
8.参考文献 33
9.附录：C++/C 代码审查表 34

火龙果 · 整理
uml.org.cn

嵌入式系统中的 C 语言编程规范

Who has seen the wind?

Neither you nor I.

When the trees bend down their heads,

The wind is passing by.

1．引言

1.1．嵌入式系统的定义

嵌入式系统（Embedded System）无疑是当前最热门、最有发展前途的 IT 应用领域之一。嵌入式系统是“嵌入到对象体系中的专用计算机系统，‘嵌入性’、‘专用性’和‘计算机系统’是嵌入式系统的三个基本要素”

嵌入式系统有多种多样的定义，但本质是相同的。美国 CMP Books 出版的 Jack Ganssle 和 Michael Barr 著作《Embedded System Dictionary》对嵌入式系统的定义为：一种计算机硬件和软件的组合，也许还有机械装置，用于实现一个特定功能。在某些特定情况下，嵌入式系统是一个大系统或产品的一部分。而国内的一些学者认为：嵌入式系统是指以应用为中心，以计算机技术为基础，软硬件可裁剪，适应应用系统对功能、可靠性、成本、体积和功耗严格要求的专门计算机系统。

1.2．本文档的目的

故事 1：

小学二年级学过一篇课文，叫做《天鹅、梭子鱼和虾》，说的是这三个动物要拉一辆车，天鹅往天上拉，梭子鱼往水里拉，而虾则拱起身子，憋红了脸往后拉，虽然三种动物拼尽了全力，但车子一动也不动。

故事 2：

90 年代初期，中国人在电脑上使用的 WPS 编辑系统，全部代码都是求伯君先生一个人用汇编写成。在那个计算机还是稀罕物的年代，也是个人英雄辈出的年代。随着 WPS 的使用，若干问题暴露了出来，但求先生却没有再对其进行完善，不是不想完善，而是完善的成本太高了！随着电脑软件的复杂性逐渐增加，由个人完成大型软件的开发已经不可能了，微软公司推出的办公软件都是由几百人来共同完成的。当前金山公司推出的 WPS 版本，也是由上百人的程序员队伍完成的。

火龙果 · 整理
uml.org.cn

第 1 个故事说明：只有个人的努力，没有统一的指挥，形不成合力，结果成事不足败事有余。正如一个团队，尽管参与工作的人员学习态度端正，工作认真努力，也许个人水平很高，但如果没有规矩，思想不统一，整个团队就没有战斗力。第 2 个故事使我想起了《三国演义》中的关羽，英勇神武，五虎上将之首，匹马斩颜良，单刀会鲁肃，好像一个人就可以扭转战局。这正如产品开发中的高手，单枪匹马可以把产品开发的所有工作都搞掂！但是，这样的“高手”，要么融入集体，带领着团队争取更大的成就，由个人英雄主义转变为集体英雄主义。在这个转变的过程，必然要去除骄傲自满，大力提倡合作精神，并且要放低身段，海纳百川。否则，“水至清则无鱼，人至察则无徒”，要么最终把自己累死，要么裹足不前成为井底之蛙。想想看，面对东吴的千军万马，曾经过五关斩六将的关云长为何会败走麦城？

对于一个团队来说，个体水平很重要，团队合作水平更重要。要提高团队合作水平，一方面要提高个体水平，另一方面要制定大家统一遵守的规则，三大纪律八项注意，步调一致才能得胜利！

当前项目组成员中，研究生占了较大比重。绝大多数研究生态度认真，工作努力，然而由于经验不足，遇到新的课题往往不知如何下手，下手后又不知如何深入。正如一个不懂水性的游泳者，被人硬推下水，扑腾了几下学会了游泳，但姿势仅限于狗刨（正如我的泳姿）。也许经过多日的扑腾，能游得比较快，获得了一些进步，但花费的时间和精力与所取得的成绩相比不成比例（正如我现在的游泳现状，凭蛮力可以与人一较高下，由于泳姿不好，遇到高手就歇菜了！）。绝大多数研究生为非计算机专业人员，未受过专业训练，软件编程随意性大，风格不统一，再加之项目组之前未加强制性规定，结果导致软件的可读性和可移植性差，软件质量不高。一些软件代码在间隔一段时间后，甚至成为了连编程者都很难看懂和修改的“一次性代码”。

为了让研究生从开始便养成良好的编程风格，以提高整个团队的生产力，在参考国内外文档的基础上，并结合多年来搞嵌入式开发的一些经验教训，编写了此文档，以规范团队的编程风格，提高软件质量。“百年大计，质量为本”。在某些情况下，虽然讲究质量会使编程者的编程速度放缓，但从整个项目的角度来看是值得的！想一想龟兔赛跑中为什么乌龟会获胜？“Slow and steady wins the race！”。

本规范主要针对单片机 C 编程语言而言，包括文件结构、命名规则、程序板式、表达式和基本语句、常量、函数设计等内容。但由于当前的编程环境大多是 C++/C 兼容的，所以也部分的涉猎了 C++的内容。

2.文件结构

每个 C 程序通常分为两个文件。一个文件用于保存程序的声明（declaration），称为头文件。另一个文件用于保存程序的实现（implementation），称为定义（definition）文件。

C 程序的头文件以“.h”为后缀，C 程序的定义文件以“.c”为后缀。

2.1. 版权和版本的声明

版权和版本的声明位于头文件和定义文件的开头，主要内容有：

（1）版权信息。

（2）文件名称，标识符，摘要。

（3）当前版本号，作者/修改者，完成日期。

（4）版本历史信息。

示例 1　版权和版本的声明

```
/*
* Copyright (c) 2012，中国海洋大学工程学院
* All rights reserved.
*
* 文件名称：filename.h
* 文件标识：
* 摘　　要：简要描述本文件的内容
*
* 当前版本：1.1
* 作　　者：输入作者（或修改者）名字
* 完成日期：2012年2月29日
*
* 取代版本：1.0
* 原作者　：输入原作者（或修改者）名字
* 完成日期：2012年1月7日
*/
```

2.2. 头文件的结构

头文件由三部分内容组成：

（1）头文件开头处的版权和版本声明（参见示例 1）。

（2）预处理块。

（3）函数和类结构声明等。

假设头文件名称为 graphics.h，头文件的结构参见示例 2。

【规则 1】为了防止头文件被重复引用，应当用 ifndef/define/endif 结构产生预处理块。

【规则 2】用 #include <filename.h> 格式来引用标准库的头文件（编译器将从标准库目

录开始搜索）。

【规则 3】用 #include “filename.h” 格式来引用非标准库的头文件（编译器将从用户的工作目录开始搜索）。

【建议 1】头文件中只存放“声明”而不存放“定义”

【建议 2】不提倡使用全局变量，尽量不要在头文件中出现象 extern int value 这类声明。

示例 2 头文件的结构

```
    // 版权和版本声明见有关示例，此处省略。

    #ifndef   GRAPHICS_H  // 防止 graphics.h 被重复引用
    #define   GRAPHICS_H

    #include <math.h>     // 引用标准库的头文件
    …
    #include “myheader.h”    // 引用非标准库的头文件
    …
    void Function1(…);    // 全局函数声明
    …
    class Box             // 类结构声明
    {
    …
    };
#endif
```

2.3 定义文件的结构

定义文件有三部分内容：

（1）定义文件开头处的版权和版本声明（参见示例 1）。

（2）对一些头文件的引用。

（3）程序的实现体（包括数据和代码）。

假设定义文件的名称为 graphics.cpp，定义文件的结构参见示例 3。

示例 3 定义文件的结构

```
    // 版权和版本声明见示例 1，此处省略。

    #include “graphics.h”    // 引用头文件
    …

    // 全局函数的实现体
    void Function1(…)
    {
  …
    }

  …
}
```

2.4. 头文件的作用

早期的编程语言如 Basic、Fortran 没有头文件的概念，C++/C 语言的初学者虽然会使用头文件，但常常不明其理。这里对头文件的作用略作解释：

（1）通过头文件来调用库功能。在很多场合，源代码不便（或不准）向用户公布，只要向用户提供头文件和二进制的库即可。用户只需要按照头文件中的接口声明来调用库功能，而不必关心接口怎么实现的。编译器会从库中提取相应的代码。

（2）头文件能加强类型安全检查。如果某个接口被实现或被使用时，其方式与头文件中的声明不一致，编译器就会指出错误，这一简单的规则能大大减轻程序员调试、改错的负担。

（3）头文件可以提高程序的可读性（清晰性）

2.5.从编译器的原理出发理解.C 与.H 文件

要理解 C 文件与头文件（即.h）有什么不同之处，首先需要弄明白编译器的工作过程，一般说来编译器会做以下几个过程：

1.预处理阶段

2.词法与语法分析阶段

3.编译阶段，首先编译成纯汇编语句，再将之汇编成跟 CPU 相关的二进制码，生成各个目标文件 (.obj 文件)

4.连接阶段，将各个目标文件中的各段代码进行绝对地址定位，生成跟特定平台相关的

可执行文件，当然，最后还可以用 objcopy 生成纯二进制码，也就是去掉了文件格式信息。（生成.exe 文件）

编译器在编译时是以 C 文件为单位进行的，也就是说如果你的项目中一个 C 文件都没有，那么你的项目将无法编译，连接器是以目标文件为单位，它将一个或多个目标文件进行函数与变量的重定位，生成最终的可执行文件，在 PC 上的程序开发，一般都有一个 main 函数，这是各个编译器的约定。

为了生成一个最终的可执行文件，就需要一些目标文件，也就是需要 C 文件，而这些 C 文件中又需要一个 main 函数作为可执行程序的入口，那么我们就从一个 C 文件入手，假定这个 C 文件内容如下：

```
    // 版权和版本声明见示例 1，此处省略。

#include "mytest.h"  //引用头文件
int main(int argc,char **argv)
{
test = 25;
printf("test................%d\n",test);
}
```

头文件内容如下：

```
    // 版权和版本声明见示例 1，此处省略。

int test;
```

现在以这个例子来讲解编译器的工作：

1.预处理阶段：编译器以 C 文件作为一 个单元，首先读这个 C 文件，发现第一句与第二句是包含一个头文件，就会在所有搜索路径中寻找这两个文件，找到之后，就会将相应头文件中再去处理宏，变量， 函数声明，嵌套的头文件包含等，检测依赖关系，进行宏替换，看是否有重复定义与声明的情况发生，最后将那些文件中所有的东东全部扫描进这个当前的 C 文件 中，形成一个中间“C 文件”

2.编译阶段，在上一步中相当于将那个头文件中的 test 变量扫描进了一个中 间 C 文件，那么 test 变量就变成了这个文件中的一个全局变量，此时就将所有这个中间 C 文件的所有变量，函数分配空间，将各个函数编译成二进制码，按照特定目标文件格式生成目标文件，在这种格式的目标文件中进行各个全局变量，函数的符号描述，将这些二进制码按照一定的标准组织成一个目标文件

3.连接阶段，将上一步生成的各个目标文件，根据一些参数，连接生成最终的可执行文件，主要的工作就是重定位各个目标文件的函数，变量等，相当于将目标文件中的二进制码

按一定的规范合到一个文件中。

再回到 C 文件与头文件各写什么内容的话题上：理论上来说 C 文件与头文件里的内容，只要是 C 语言所支持的，无论写什么都可以的，比如你在头文件中写函数体，只要在任何一个 C 文件包含此头文件就可以将这个函数编译成目标文件的一部分（编译是以 C 文件为单位的，如果不在任何 C 文件中包含此头文件的话，这段代码就形同虚设），你可以在 C 文件中进行函数声明，变量声明，结构体声明，这也不成问题!!! 那为何一定要分成头文件与 C 文件呢？又为何一般都在头件中进行函数，变量声明，宏声明，结构体声明 呢？而在 C 文件中去进行变量定义，函数实现呢？？原因如下：

1.如果在头文件中实现一个函数体，那么如果在多个 C 文件中引用它，而且又同时编译多个 C 文件，将其生成的目标文件连接成一个可执行文件，在每个引用此头文件的 C 文件所生成的目标文件中，都有一份这个函数的代码，如果这段函数又没有定义成局部函数，那么在连接时，就会发现多个相同的函数，就会报错

2.如果在头文件中定义全局变量，并且将此全局变量赋初值，那么在多个引用此头文件的 C 文件中同样存在相同变量名的拷贝，关键是此变量被赋了初值，所以编译器就会将此变量放入 DATA 段，最终在连接阶段，会在 DATA 段中存在多个相同的变量，它无法将这些变量统一成一个变量，也就是仅为此变量分配一个空间，而不是多份空间，假定这个变量在头文件没有赋初值，编译器就会将之放入 BSS 段，连接器会对 BSS 段的多个同名变量仅分配一个存储空间

3.如果在 C 文件中声明宏，结构体，函数等，那么我要在另一个 C 文件中引用相应的宏，结构体，就必须再做一次重复的工作，如果我改了一个 C 文件中的一个声明，那么又忘了改其它 C 文件中的声明，这不就出了大问题了，程序的逻辑就变成了你不可想象的了，如果把这些公共的东东放在一个头文件中，想用它的 C 文件就只需要引用一个就 OK 了!!! 这样岂不方便，要改某个声明的时候，只需要动一下头文件就行了

4.在头文件中声明结构体，函数等，当你需要将你的代码封装成一个库，让别人来用你的代码，你又不想公布源码，那么人家如何利用你的库呢？也就是如何利用你的库中的各个函数呢？？一种方法是公布源码，别人想怎么用就怎么用，另一种是提供头文件，别人从头文件中看你的函数原型，这样人家才知道如何调用你写的函数，就如同你调用 printf 函数一样，里面的参数是怎样的？？你是怎么知道的？？还不是看人家的头文件中的相关声明啊!!! 当然这些东东都成了 C 标准，就算不看人家的头文件，你一样可以知道怎么使用。

2.6 目录结构

如果一个软件的头文件数目比较多（如超过十个），通常应将头文件和定义文件分别保存于不同的目录，以便于维护。

例如可将头文件保存于 include 目录，将定义文件保存于 source 目录（可以是多级目录）。

如果某些头文件是私有的，它不会被用户的程序直接引用，则没有必要公开其“声明”。为了加强信息隐藏，这些私有的头文件可以和定义文件存放于同一个目录。

3. 命名规则

比较著名的命名规则当推 Microsoft 公司的“匈牙利”法，该命名规则的主要思想是“在变量和函数名中加入前缀以增进人们对程序的理解”。例如所有的字符变量均以 ch 为前缀，若是指针变量则追加前缀 p。如果一个变量由 ppch 开头，则表明它是指向字符指针的指针。

“匈牙利”法最大的缺点是烦琐，例如

```
int    i, j, k;
float  x, y, z;
```

倘若采用“匈牙利”命名规则，则应当写成

```
int    iI, iJ, ik;  // 前缀 i 表示 int 类型
float  fX, fY, fZ;  // 前缀 f 表示 float 类型
```

如此烦琐的程序会让绝大多数程序员无法忍受。

据考察，没有一种命名规则可以让所有的程序员赞同，程序设计教科书一般都不指定命名规则。命名规则对软件产品而言并不是“成败悠关”的事，我们不要化太多精力试图发明世界上最好的命名规则，而应当制定一种令大多数项目成员满意的命名规则，并在项目中贯彻实施。

3.1 共性规则

共性规则指大多数情况下必须遵循的规则，如果有本规定没有涉及到的规则，可以根据本规则进行类推。

【规则 1】应当直观且可以拼读，可望文知意，不必进行“解码”。

标识符最好采用英文单词或其组合，便于记忆和阅读。切忌使用汉语拼音来命名。程序中的英文单词一般不会太复杂，用词应当准确。例如不要把 CurrentValue 写成 NowValue。

【规则 2】长度应当符合“min-length && max-information”原则。

几十年前老 ANSI C 规定名字不准超过 6 个字符，现今的 C 语言不再有此限制。一般来说，长名字能更好地表达含义，所以函数名、变量名、类名长达十几个字符不足为怪。那么名字是否越长约好？不见得！例如变量名 MaxVal 就比 MaxValueUntilOverflow 好用。单字符的名字也是有用的，常见的如 i,j,k,m,n,x,y,z 等，它们通常可用作函数内的局部变量。

【规则 3】 程序中不要出现仅靠大小写区分的相似的标识符。

例如：

```
int  x,  X;      // 变量 x 与 X 容易混淆
void foo(int x);// 函数 foo 与 FOO 容易混淆
void FOO(float x);
```

【规则 4】程序中不要出现标识符完全相同的局部变量和全局变量，尽管两者的作用域不同而不会发生语法错误，但会使人误解。（注：一个较好风格的程序，应该尽量不用或少用全局变量，以增加程序的可移植性）

【规则 5】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：

```
int MinValue;
int MaxValue;

int SetValue(…);
int GetValue(…);
```

【规则 6】尽量避免名字中出现数字编号，如 Value1,Value2 等，除非逻辑上的确需要编号。这是为了防止程序员偷懒，不肯为命名动脑筋而导致产生无意义的名字（因为用数字编号最省事）。

【规则 7】为了代码复用，命名中应避免使用与具体项目相关的前缀。

3.2 扩展规则

扩展规则是在遵守共性规则的基础上，对相关项目进行的更为明确的规定。如果有和共性规则冲突的，以扩展规则为准。在扩展规则没有规定的项目，可以根据共性规则类推处理。

【规则 1】预定义(#define)：只使用大写字母，下划线和数字来表示。

例如：

```
#define        MAX_LENGTH          1
```

【规则 2】宏和常量命名：只使用大写字母，下划线和数字。宏和常量全部用大写字母来命名，词与词之间用下划线分隔。对程序中用到的数字均应用有意义的枚举或宏来代替。

例如：

```
const int MAX = 100;
const int MAX_LENGTH = 100;
```

【规则 3】变量命名规则： <<范围>_类型>名称。

范围	前缀
结构体的成员变量	m_
静态变量	s_
全局变量	g_
局部变量	无前缀

类型	前缀
指针（Pointer）	p
枚举（Enumeration）	e
结构（Structure）	st
布尔（Boolean）	b
浮点（Float）	f
双精度（Double）	d
字符（Char）	c
其他数字类型, e.g. byte, (unsigned) int, (unsigned)long int, (unsigned)short, (unsigned) short int, (unsigned) long long	n

例如：

```
//全局变量
int g_nMaxCount;
//函数体内局部变量
MyEnumType    eParsingMode;
```

局部循环体控制变量优先使用 i、j、k 等；局部长度变量优先使用 len、num 等；临时中间变量优先使用 temp、tmp 等。

【规则 4】结构和类型定义（typedef）

按 Camel-Style 方式命名，如 ThisIsAnExampleStructOrEnumOrTyedef 。避免使用下

划线。

【规则 5】枚举：按 Camel-Style 方式命名，如 ThisIsAnExampleStructOrEnumOrTyedef 。避免使用下划线。

```
Enum DataSetState
{
ValidModified = 1,
InvalidModified = 2,
AllDataLoaded = 3
}
```

【规则 6】函数名按 Camel-Style 方式命名

例如：

```
void  Draw(void);            // 函数名
void  SetValue(int value);   // 函数名
```

3.3 嵌入式系统的数据类型命名

这个问题在嵌入式系统中很重要！说他重要，是因为嵌入式系统不像通用型系统，例如 PC 机，由于操作系统、编程平台的统一，long，short，int，char 等类型的含义都是高度一致的，每种类型包括几个字节都是确定的，因此把这个问题给掩盖了。而嵌入式系统恰恰相反，你所面对的硬件平台是多样的，例如 8 位机、16 位机、32 位机；面对的软件平台也是多样的，例如不同厂家的开发系统差别很大，例如 MSP430 使用的 TI 的 IDE，而 STM32 使用的是 KEIL/IAR 系统，STM8 虽然也是意法半导体的产品，但使用的开发环境也是不同的。这样就会带来一个问题，举例说明："在 STM32 的开发环境和 STM8 的开发环境下，short 类型代表的字节数是否相同呢？

要弄明白这个问题，我们先来看看百度百科上的有关"short"类型的解释：

"C 语言中，short 是定义一种整型变量家族的一种。例如 short i；表示定义一个短整型的变量 i。依据程序编译器的不同 short 定义的字节数不同。标准定义 short 短整型变量不得低于 16 位，即两个字节。编译器头文件夹里面的 limits.h 定义了 short 能表示的大小：SHRT_MIN~SHRT_MAX。在 32 位平台下如 windows(32 位)中 short 一般为 16 位。"

我们再来看一下百度名片上对"char"类型的解释：

"char 是 C/C++整型数据中比较古怪的一个，其它的如 int/long/short 等不指定 signed/unsigned 时都默认是 signed，但 char 在标准中是 unsigned，编译器可以实现为带符号的，也可以实现为不带符号的，有些编译器如 pSOS 的编译器，还可以通过编译开关来指定它是有符号数还是无符号数。 "

不需要再过多的举例子，我们就知道问题有多严重！我曾经让研究生将一个在 STM32 下实现的仪表，改造成 STM8S 下来实现，结果发现运算结果总是不对，特别是函数运算的结果差别很大，根本无法使用！而该函数在 STM32 开发环境下非常稳定。同样的错误也出现在 MSP430 向 STM8S 的移植中。

为此，需要对采取措施，以避免这种情况的出现。在实践的过程中，感觉这种定义法比较合适，举例如下：

```
/*!< Signed integer types    */
typedef     signed char        int8_t;
typedef     signed short       int16_t;
typedef     signed long        int32_t;

/*!< Unsigned integer types    */
typedef unsigned char        uint8_t;
typedef unsigned short       uint16_t;
typedef unsigned long        uint32_t;

/*!< STM8 Standard Peripheral Library old types (maintained for legacy purpose) */

typedef   int32_t        s32;
typedef   int16_t        s16;
typedef   int8_t         s8;

typedef   uint32_t       u32;
typedef   uint16_t       u16;
typedef   uint8_t        u8;
```

这样，在定义变量的情况下，统一用 s32/s16/s8 来定义有符号数，u32/u16/u8 定义无符号数，超出 32 位的数据，就用 float 来定义，基本上可以达到比较通用的目的。

4．程序的版式

版式虽然不会影响程序的功能，但会影响可读性。程序的版式追求清晰、美观，是程序风格的重要构成因素。可以把程序的版式比喻为“书法”。好的“书法”可让人对程序一目了然，看得兴致勃勃。

4.1．空行

空行起着分隔程序段落的作用。空行得体（不过多也不过少）将使程序的布局更加清晰。空行不会浪费内存，虽然打印含有空行的程序是会多消耗一些纸张，但是值得。所以不要舍不得用空行。

【规则 1】在每个函数定义结束之后都要加空行。

【规则 2】在一个函数体内，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。

(a) 函数之间的空行	(b) 函数内部的空行

// 空行
void Function1(…)
{
…
}
// 空行
void Function2(…)
{
…
}
// 空行
void Function3(…)
{
…
}
	// 空行
	while (condition)
	{
	statement1;
	// 空行
	if (condition)
	{
	statement2;
	}
	else
	{
	statement3;
	}
	// 空行
	statement4;
	}

4.2. 代码行

【规则 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。

【规则 2】if、for、while、do 等语句自占一行，执行语句不得紧跟其后。不论执行语句有多少都要加{}。这样可以防止书写失误。

代码行书写示例

风格良好的代码行	**风格不良**的代码行
int width;　　// 宽度 int height;　　// 高度 int depth;　　// 深度	int width, height, depth; // 宽度高度深度

x = a + b;
y = c + d;
z = e + f;
	X = a + b;　y = c + d;　z = e + f;

if (width < height)
{
dosomething();
}
	if (width < height) dosomething();

for (initialization; condition; update)
{
dosomething();
}
// 空行
other();
	for (initialization; condition; update)
	dosomething();
	other();

【建议 1】尽可能在定义变量的同时初始化该变量（就近原则）

如果变量的引用处和其定义处相隔比较远，变量的初始化很容易被忘记。如果引用了未被初始化的变量，可能会导致程序错误。本建议可以减少隐患。例如

```
int width = 10;      // 定义并初给化 width
int height = 10;     // 定义并初给化 height
int depth = 10;      // 定义并初给化 depth
```

4.3. 代码行内的空格

【规则 1】关键字之后要留空格。象 const、virtual、inline、case 等关键字之后至少要留一个空格，否则无法辨析关键字。象 if、for、while 等关键字之后应留一个空格再跟左括号‘(’，以突出关键字。

【规则 2】函数名之后不要留空格，紧跟左括号‘(’，以与关键字区别。

【规则 3】‘(’ 向后紧跟，‘)’、‘,’、‘;’ 向前紧跟，紧跟处不留空格。

【规则 4】‘,’ 之后要留空格，如 Function(x, y, z)。如果‘;’ 不是一行的结束符号，其后要留空格，如 for (initialization; condition; update)。

【规则 5】赋值操作符、比较操作符、算术操作符、逻辑操作符、位域操作符，如“=”、“+=”“>=”、“<=”、“+”、“*”、“%”、“&&”、“||”、“<<”，“^”等二元操作符的前后应当加空格。

【规则 6】一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。

【规则 7】象“[ ]”、“.”、“->”这类操作符前后不加空格。

【建议 1】对于表达式比较长的 for 语句和 if 语句，为了紧凑起见可以适当地去掉一些空格，如 for (i=0; i<10; i++)和 if ((a<=b) && (c<=d))

代码行内的空格说明	
void Func1(int x, int y, int z);	// 良好的风格
void Func1 (int x,int y,int z);	// 不良的风格
if (year >= 2000)	// 良好的风格
if(year>=2000)	// 不良的风格
if ((a>=b) && (c<=d))	// 良好的风格
if(a>=b&&c<=d)	// 不良的风格
for (i=0; i<10; i++)	// 良好的风格
for(i=0;i<10;i++)	// 不良的风格
for (i = 0; I < 10; i ++)	// 过多的空格
x = a < b ? a : b;	// 良好的风格
x=a<b?a:b;	// 不好的风格
int *x = &y;	// 良好的风格
int * x = & y;	// 不良的风格
array[5] = 0;	// 不要写成 array [5] = 0;
a.Function();	// 不要写成 a . Function();
b->Function();	// 不要写成 b -> Function();

4.4. 对齐

【规则 1】程序的分界符‘{’和‘}’应独占一行并且位于同一列，同时与引用它们的语句左对齐。

【规则 2】{ }之内的代码块在‘{’右边数格处左对齐。

风格良好的对齐	风格不良的对齐

void Function(int x)
{
… // program code
}
	void Function(int x){
	… // program code
	}

if (condition)
{
… // program code
}
else
{
… // program code
}
	if (condition){
	… // program code
	}
	else {
	… // program code
	}

for (initialization; condition; update)
{
… // program code
}
	for (initialization; condition; update){
	… // program code
	}

While (condition)
{
… // program code
}
	while (condition){
	… // program code
	}

如果出现嵌套的｛｝，则使用缩进对齐，如：
{
…
{
…
}
…
}	

4.5. 长行拆分

【规则 1】代码行最大长度宜控制在 70 至 80 个字符以内。代码行不要过长，否则眼睛看不过来，也不便于打印。

【规则 2】长表达式要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操

作符)。拆分出的新行要进行适当的缩进，使排版整齐，语句可读。

```
if ((very_longer_variable1 >= very_longer_variable12)
&& (very_longer_variable3 <= very_longer_variable14)
&& (very_longer_variable5 <= very_longer_variable16))
{
    dosomething();
}
```

```
virtual CMatrix CMultiplyMatrix (CMatrix leftMatrix,
                                 CMatrix rightMatrix);
```

```
for (very_longer_initialization;
     very_longer_condition;
     very_longer_update)
{
    dosomething();
}
```

4.6. 修饰符的位置

修饰符 * 和 & 应该靠近数据类型还是该靠近变量名，是个有争议的活题。

若将修饰符 * 靠近数据类型，例如：int* x; 从语义上讲此写法比较直观，即 x 是 int 类型的指针。

上述写法的弊端是容易引起误解，例如：int* x, y; 此处 y 容易被误解为指针变量。虽然将 x 和 y 分行定义可以避免误解，但并不是人人都愿意这样做。

【规则 1】应当将修饰符 * 和 & 紧靠变量名

例如：

```
char  *name;
      int   *x, y;     // 此处 y 不会被误解为指针
```

4.7. 注释

C 语言的注释符为“/*…*/”。C++语言中，程序块的注释常采用“/*…*/”，行注释一般采用“//…”。注释通常用于：

（1）版本、版权声明；

（2）函数接口说明；

（3）重要的代码行或段落提示。

虽然注释有助于理解代码，但注意不可过多地使用注释。

【规则 1】注释是对代码的“提示”，而不是文档。程序中的注释不可喧宾夺主，注释太多了会让人眼花缭乱。注释的花样要少。

【规则 2】如果代码本来就是清楚的，则不必加注释。否则多此一举，令人厌烦。例如

```
i++;    // i 加 1，多余的注释
```

【规则 3】边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。

【规则 4】注释应当准确、易懂，防止注释有二义性。错误的注释不但无益反而有害。

【规则 5】尽量避免在注释中使用缩写，特别是不常用缩写。

【规则 6】注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方。

【规则 7】当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处加注释，便于阅读。

/*
* 函数介绍：
* 输入参数：
* 输出参数：
* 返回值　：
*/
void Function(float x, float y, float z)
{
…
}
	if (…)
	{
	…
	while (…)
	{
	…
	} // end of while
	…
	} // end of if

4.8. 使用的端口说明

嵌入式系统工程中应该包括一个系统端口说明文件，最好是 TXT 文件，说明在系统使用到的单片机端口用途，以 AMT 软件为例：

```
/*************************************************************************
模拟输入端口说明：
    STM32F103中的PA.00-PA.07 和 PC.00-PC.02 均为模拟输入引脚：
```

```
PA.00 is XD_S,       档位选择传感器
PA.01 is YOUYA_S,    压力传感器
PA.02 is H_SHIFT,    高传动比信息
PA.03 is WENDING,    稳定位置信息
PA.04 is L_SHIFT,    低传动比信息
PA.05 is CLUTCH,     换到空挡的信息
PA.06 is BRAKE,      脚刹连接器
PA.07 is DOOR,       门的连接器
PC.00 is LOW_POWER,  启动电源
PC.01 is HD_S,       啮合位置传感器
PC.02 is LH_S,       离合器位置传感器
模拟输出端口说明：
……
数字输入端口说明：
……
数字输出端口说明：
……
通讯端口说明：
    串口 1：
    串口 2：
    CAN 总线接口：
    ……
*************************************************************************
***/
```

5. 表达式和基本语句

表达式和语句都属于 C 的短语结构语法。它们看似简单，但使用时隐患比较多。本节归纳了正确使用表达式和语句的一些规则与建议。

5.1. 运算符的优先级

C 语言的运算符有数十个，运算符的优先级与结合律如下表所示。注意一元运算符 + - * 的优先级高于对应的二元运算符。

运算符的优先级与结合律

优先级	运算符	结合律

从高到低排列	( ) [ ] -> .	从左至右
	! ~ ++ -- （类型） sizeof + - * &	**从右至左**
	* / %	从左至右
	+ -	从左至右
	<< >>	从左至右
	< <= > >=	从左至右
	== !=	从左至右
	&	从左至右
	^	从左至右
	|	从左至右
	&&	从左至右
	||	**从右至左**
	?:	**从右至左**
	= += -= *= /= %= &= ^= |= <<= >>=	从左至右

【规则 1】如果代码行中的运算符比较多，用括号确定表达式的操作顺序，避免使用默认的优先级。

由于将表中的优先级熟记是比较困难的，为了防止产生歧义并提高可读性，应当用括号确定表达式的操作顺序。例如：

```
word = (high << 8) | low
if ((a | b) && (a & c))
```

5.2. 复合表达式

如 a = b = c = 0 这样的表达式称为复合表达式。允许复合表达式存在的理由是：

（1）书写简洁；

（2）可以提高编译效率。

但要防止滥用复合表达式。

【规则 1】不要编写太复杂的复合表达式。

例如：

```
i = a >= b && c < d && c + f <= g + h ; // 复合表达式过于复杂
```

【规则 2】不要有多用途的复合表达式。

例如：

```
d = (a = b + c) + r ;
```

该表达式既求 a 值又求 d 值。应该拆分为两个独立的语句：

```
a = b + c;
d = a + r;
```

【规则 3】不要把程序中的复合表达式与“真正的数学表达式”混淆。

例如：

```
if (a < b < c)          // a < b < c 是数学表达式而不是程序表达式
```

并不表示

```
if ((a<b) && (b<c))
```

而是成了令人费解的

```
if ( (a<b)<c )
```

5.3. if 语句

if 语句是 C++/C 语言中最简单、最常用的语句，然而很多程序员用隐含错误的方式写 if 语句。本节以“与零值比较”为例，展开讨论。

5.3.1. 布尔变量与零值比较

【规则 1】不可将布尔变量直接与 TRUE、FALSE 或者 1、0 进行比较。

根据布尔类型的语义，零值为“假”（记为 FALSE），任何非零值都是“真”（记为 TRUE）。TRUE 的值究竟是什么并没有统一的标准。例如 Visual C++ 将 TRUE 定义为 1，而 Visual Basic 则将 TRUE 定义为-1。

假设布尔变量名字为 flag，它与零值比较的标准 if 语句如下：

```
if (flag)    // 表示 flag 为真
if (!flag)   // 表示 flag 为假
```

其它的用法都属于不良风格，例如：

```
if (flag == TRUE)
if (flag == 1 )
if (flag == FALSE)
if (flag == 0)
```

5.3.2. 整型变量与零值比较

【规则 2】应当将整型变量用“==”或“！=”直接与 0 比较。

假设整型变量的名字为 value，它与零值比较的标准 if 语句如下：

```
if (value == 0)
if (value != 0)
```

不可模仿布尔变量的风格而写成

```
if (value)      // 会让人误解 value 是布尔变量
if (!value)
```

在上述的比较中，有不少有经验的程序员会写成：

```
if (0 == value)
```

为什么要这样写呢？这是为了避免笔误，例如把“==”写成了“=”，本来是比较是否相等，结果成了赋值语句，编译器却发现不了这样的错误。

5.3.3. 浮点变量与零值比较

【规则 3】不可将浮点变量用“==”或“！=”与任何数字比较。

千万要留意，无论是 float 还是 double 类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”形式。

假设浮点变量的名字为 x，应当将

```
if (x == 0.0)    // 隐含错误的比较
```

转化为

```
if ((x>=-EPSINON) && (x<=EPSINON))
```

其中 EPSINON 是允许的误差（即精度）。

5.3.4. 指针变量与零值比较

【规则 4】应当将指针变量用“==”或“！=”与 NULL 比较。

指针变量的零值是“空”（记为 NULL）。尽管 NULL 的值与 0 相同，但是两者意义不同。假设指针变量的名字为 p，它与零值比较的标准 if 语句如下：

```
if (p == NULL)  // p 与 NULL 显式比较，强调 p 是指针变量
if (p != NULL)
```

不要写成

```
if (p == 0)     // 容易让人误解 p 是整型变量
if (p != 0)
```

或者

```
if (p)          // 容易让人误解 p 是布尔变量
if (!p)
```

5.3.5. 对 if 语句的补充说明

有时候我们可能会看到 if (NULL == p) 这样古怪的格式。不是程序写错了，是程序员为了防止将 if (p == NULL) 误写成 if (p = NULL)，而有意把 p 和 NULL 颠倒。编译器认为 if (p = NULL) 是合法的，但是会指出 if (NULL = p)是错误的，因为 NULL 不能被赋值。

程序中有时会遇到 if/else/return 的组合，应该将如下不良风格的程序

```
        if (condition)
            return x;
        return y;
```

改写为

```
        if (condition)
        {
            return x;
        }
        else
        {
        return y;
    }
```

或者改写成更加简练的

```
    return (condition ? x : y);
```

5.4. 循环语句的效率

C++/C 循环语句中，for 语句使用频率最高，while 语句其次，do 语句很少用。本节重点论述循环体的效率。提高循环体效率的基本办法是降低循环体的复杂性。

【建议 1】在多重循环中，如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少 CPU 跨切循环层的次数。例如示例 6-4-4(b)的效率比示例 6-4-4(a)的高。

for (row=0; row<100; row++)
{
for (col=0; col<5; col++)
{
sum = sum + a[row][col];
}
	for (col=0; col<5; col++)
	{
	for (row=0; row<100; row++)
	{
	sum = sum + a[row][col];
	}

}	}
示例 (a) 低效率：长循环在最外层	示例 (b) 高效率：长循环在最内层

【建议 2】如果循环体内存在逻辑判断，并且循环次数很大，宜将逻辑判断移到循环体的外面。下面的示例 (c)的程序比示例 (d)多执行了 N-1 次逻辑判断。并且由于前者老要进行逻辑判断，打断了循环“流水线”作业，使得编译器不能对循环进行优化处理，降低了效率。如果 N 非常大，最好采用示例 6-4-4(d)的写法，可以提高效率。如果 N 非常小，两者效率差别并不明显，采用示例 6-4-4(c)的写法比较好，因为程序更加简洁。

for (i=0; i<N; i++)
{
if (condition)
DoSomething();
else
DoOtherthing();
}
	if (condition)
	{
	for (i=0; i<N; i++)
	DoSomething();
	}
	else
	{
	for (i=0; i<N; i++)
	DoOtherthing();
	}

表 6-4-4(c) 效率低但程序简洁	表 6-4-4(d) 效率高但程序不简洁

5.5. for 语句的循环控制变量

【规则 1】不可在 for 循环体内修改循环变量，防止 for 循环失去控制。

【建议 2】建议 for 语句的循环控制变量的取值采用“半开半闭区间”写法。

示例 (a)中的 x 值属于半开半闭区间“0 =< x < N”，起点到终点的间隔为 N，循环次数为 N。

示例 (b)中的 x 值属于闭区间“0 =< x <= N-1”，起点到终点的间隔为 N-1，循环次数为 N。

相比之下，示例 (a)的写法更加直观，尽管两者的功能是相同的。

for (int x=0; x<N; x++)
{
…
}
	for (int x=0; x<=N-1; x++)
	{
	…
	}

示例 6-4-5(a) 循环变量属于半开半闭区间	示例 6-4-5(b) 循环变量属于闭区间

5.6. switch 语句

有了 if 语句为什么还要 switch 语句?

switch 是多分支选择语句，而 if 语句只有两个分支可供选择。虽然可以用嵌套的 if 语句来实现多分支选择，但那样的程序冗长难读。这是 switch 语句存在的理由。

switch 语句的基本格式是：

```
switch (variable)
{
  case value1 : …
                break;
  case value2 : …
                break;
…
    default :   …
                break;
}
```

【规则 1】每个 case 语句的结尾不要忘了加 break，否则将导致多个分支重叠（除非有意使多个分支重叠）。

【规则 2】不要忘记最后那个 default 分支。即使程序真的不需要 default 处理，也应该保留语句 default：break; 这样做并非多此一举，而是为了防止别人误以为你忘了 default 处理。

5.7. goto 语句

自从提倡结构化设计以来，goto 就成了有争议的语句。首先，由于 goto 语句可以灵活跳转，如果不加限制，它的确会破坏结构化设计风格。其次，goto 语句经常带来错误或隐患。它可能跳过了某些对象的构造、变量的初始化、重要的计算等语句，例如：

```
goto state;
String s1, s2;   // 被 goto 跳过
int sum = 0;     // 被 goto 跳过
…
```

```
state:
…
```

如果编译器不能发觉此类错误，每用一次 goto 语句都可能留下隐患。

很多人建议废除 C++/C 的 goto 语句，以绝后患。但实事求是地说，错误是程序员自己造成的，不是 goto 的过错。goto 语句至少有一处可显神通，它能从多重循环体中咻地一下子跳到外面，用不着写很多次的 break 语句；例如

```
{ …
    { …
        { …
            goto error;
        }
    }
}
error:
…
```

就象楼房着火了，来不及从楼梯一级一级往下走，可从窗口跳出火坑。所以我们主张少用、慎用 goto 语句，而不是禁用。

6.常量

常量是一种标识符，它的值在运行期间恒定不变。C 语言用 #define 来定义常量（称为宏常量）。C++ 语言除了 #define 外还可以用 const 来定义常量（称为 const 常量）。

6.1. 为什么需要常量

如果不使用常量，直接在程序中填写数字或字符串，将会有什么麻烦？

（1）程序的可读性（可理解性）变差。程序员自己会忘记那些数字或字符串是什么意思，用户则更加不知它们从何处来、表示什么。

（2）在程序的很多地方输入同样的数字或字符串，难保不发生书写错误。

（3）如果要修改数字或字符串，则会在很多地方改动，既麻烦又容易出错。

【规则 1】 尽量使用含义直观的常量来表示那些将在程序中多次出现的数字或字符串。

例如：

```
#define      MAX   100        /*  C 语言的宏常量  */
const int    MAX = 100;       //  C++ 语言的 const 常量
const float  PI = 3.14159;    //  C++ 语言的 const 常量
```

6.2. const 与 #define 的比较

由于目前嵌入式语言大多是C++/C兼容的，可以用const来定义常量，也可以用 #define 来定义常量。但是前者比后者有更多的优点：

（1）const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误（边际效应）。

（2）有些集成化的调试工具可以对 const 常量进行调试，但是不能对宏常量进行调试。

【规则 1】在 C++ 程序中只使用 const 常量而不使用宏常量，即 const 常量完全取代宏常量。

6.3 常量定义规则

【规则 1】需要对外公开的常量放在头文件中，不需要对外公开的常量放在定义文件的头部。为便于管理，可以把不同模块的常量集中存放在一个公共的头文件中。

【规则 2】如果某一常量与其它常量密切相关，应在定义中包含这种关系，而不应给出一些孤立的值。

例如：

```
const  float   RADIUS = 100;
const  float   DIAMETER = RADIUS * 2;
```

7. 函数设计

函数是 C++/C 程序的基本功能单元，其重要性不言而喻。函数设计的细微缺点很容易导致该函数被错用，所以光使函数的功能正确是不够的。本节重点论述函数的接口设计和内部实现的一些规则。

函数接口的两个要素是参数和返回值。C 语言中，函数的参数和返回值的传递方式有两种：值传递（pass by value）和指针传递（pass by pointer）。

7.1. 参数的规则

【规则 1】参数的书写要完整，不要贪图省事只写参数的类型而省略参数名字。如果函数没有参数，则用 void 填充。

例如：

```
void SetValue(int width, int height);    // 良好的风格
void SetValue(int, int);                 // 不良的风格
float GetValue(void);        // 良好的风格
float GetValue();            // 不良的风格
```

【规则 2】参数命名要恰当，顺序要合理。

例如编写字符串拷贝函数 StringCopy，它有两个参数。如果把参数名字起为 str1 和 str2，例如

```
void StringCopy(char *str1, char *str2);
```

那么我们很难搞清楚究竟是把 str1 拷贝到 str2 中，还是刚好倒过来。

可以把参数名字起得更有意义，如叫 strSource 和 strDestination。这样从名字上就可以看出应该把 strSource 拷贝到 strDestination。

还有一个问题，这两个参数那一个该在前那一个该在后？参数的顺序要遵循程序员的习惯。一般地，应将目的参数放在前面，源参数放在后面。

如果将函数声明为：

```
void StringCopy(char *strSource, char *strDestination);
```

别人在使用时可能会不假思索地写成如下形式：

```
char str[20];
StringCopy(str, “Hello World”);   // 参数顺序颠倒
```

【规则 3】如果参数是指针，且仅作输入用，则应在类型前加 const，以防止该指针在函数体内被意外修改。

例如：

```
void StringCopy(char *strDestination, const char *strSource);
```

【建议 1】避免函数有太多的参数，参数个数尽量控制在 5 个以内。如果参数太多，在使用时容易将参数类型或顺序搞错。

【建议 2】尽量不要使用类型和数目不确定的参数。

C 标准库函数 printf 是采用不确定参数的典型代表，其原型为：

```
int printf(const chat *format[, argument]…);
```

这种风格的函数在编译时丧失了严格的类型安全检查。

7.2.返回值的规则

【规则 1】不要省略返回值的类型。

C 语言中，凡不加类型说明的函数，一律自动按整型处理。这样做不会有什么好处，却容易被误解为 void 类型。

C++语言有很严格的类型安全检查，不允许上述情况发生。由于 C++程序可以调用 C 函数，为了避免混乱，规定任何 C++/ C 函数都必须有类型。如果函数没有返回值，那么应声

明为 void 类型。

【规则 2】函数名字与返回值类型在语义上不可冲突。

违反这条规则的典型代表是 C 标准库函数 getchar。

例如：

```
char c;
c = getchar();
if (c == EOF)
...
```

按照 getchar 名字的意思，将变量 c 声明为 char 类型是很自然的事情。但不幸的是 getchar 的确不是 char 类型，而是 int 类型，其原型如下：

```
int getchar(void);
```

由于 c 是 char 类型，取值范围是[-128，127]，如果宏 EOF 的值在 char 的取值范围之外，那么 if 语句将总是失败，这种“危险”人们一般哪里料得到！导致本例错误的责任并不在用户，是函数 getchar 误导了使用者。

【规则 3】不要将正常值和错误标志混在一起返回。正常值用输出参数获得，而错误标志用 return 语句返回。

回顾上例，C 标准库函数的设计者为什么要将 getchar 声明为令人迷糊的 int 类型呢？他会那么傻吗？

在正常情况下，getchar 的确返回单个字符。但如果 getchar 碰到文件结束标志或发生读错误，它必须返回一个标志 EOF。为了区别于正常的字符，只好将 EOF 定义为负数（通常为负 1）。因此函数 getchar 就成了 int 类型。

我们在实际工作中，经常会碰到上述令人为难的问题。为了避免出现误解，我们应该将正常值和错误标志分开。即：正常值用输出参数获得，而错误标志用 return 语句返回。

函数 getchar 可以改写成 BOOL GetChar(char *c);

虽然 gechar 比 GetChar 灵活，例如 putchar(getchar()); 但是如果 getchar 用错了，它的灵活性又有什么用呢？

【建议 1】有时候函数原本不需要返回值，但为了增加灵活性如支持链式表达，可以附加返回值。

例如字符串拷贝函数 strcpy 的原型：

```
char *strcpy(char *strDest，const char *strSrc);
```

strcpy 函数将 strSrc 拷贝至输出参数 strDest 中，同时函数的返回值又是 strDest。这样做并非多此一举，可以获得如下灵活性：

```
char str[20];
int  length = strlen( strcpy(str, "Hello World" ) );
```

【建议 2】如果函数的返回值是一个对象，有些场合用“引用传递”替换“值传递”可以提高效率。而有些场合只能用“值传递”而不能用“引用传递”，否则会出错。

例如：

```
class String
{…
// 赋值函数
String & operate=(const String &other);
// 相加函数，如果没有 friend 修饰则只许有一个右侧参数
friend  String   operate+( const String &s1, const String &s2);
private:
char *m_data;
}
```

String 的赋值函数 operate = 的实现如下：

```
String & String::operate=(const String &other)
{
if (this == &other)
    return *this;
delete m_data;
m_data = new char[strlen(other.data)+1];
strcpy(m_data, other.data);
return *this;   // 返回的是 *this 的引用，无需拷贝过程
}
```

对于赋值函数，应当用“引用传递”的方式返回 String 对象。如果用“值传递”的方式，虽然功能仍然正确，但由于 return 语句要把 *this 拷贝到保存返回值的外部存储单元之中，增加了不必要的开销，降低了赋值函数的效率。例如：

```
String a,b,c;
…
a = b;       // 如果用“值传递”，将产生一次 *this 拷贝
a = b = c;   // 如果用“值传递”，将产生两次 *this 拷贝
```

String 的相加函数 operate + 的实现如下：

```
String  operate+(const String &s1, const String &s2)
  {
  String temp;
  delete temp.data;   // temp.data 是仅含‘\0’的字符串
      temp.data = new char[strlen(s1.data) + strlen(s2.data) +1];
      strcpy(temp.data, s1.data);
      strcat(temp.data, s2.data);
      return temp;
  }
```

对于相加函数，应当用“值传递”的方式返回 String 对象。如果改用“引用传递”，那么函数返回值是一个指向局部对象 temp 的“引用”。由于 temp 在函数结束时被自动销毁，将导致返回的“引用”无效。例如：

```
c = a + b;
```

此时 a + b 并不返回期望值，c 什么也得不到，留下了隐患。

7.3. 函数内部实现的规则

不同功能的函数其内部实现各不相同，看起来似乎无法就“内部实现”达成一致的观点。但根据经验，我们可以在函数体的“入口处”和“出口处”从严把关，从而提高函数的质量。

【规则 1】在函数体的“入口处”，对参数的有效性进行检查。

很多程序错误是由非法参数引起的，我们应该充分理解并正确使用“断言”（assert）来防止此类错误。详见 6.6.5 节“使用断言”。

【规则 2】在函数体的“出口处”，对 return 语句的正确性和效率进行检查。

如果函数有返回值，那么函数的“出口处”是 return 语句。我们不要轻视 return 语句。如果 return 语句写得不好，函数要么出错，要么效率低下。

注意事项如下：

（1）return 语句不可返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。例如

```
char * Func(void)
{
    char str[] = “hello world”;   // str 的内存位于栈上
    …
```

```
    return str;      // 将导致错误
}
```

（2）要搞清楚返回的究竟是“值”、“指针”还是“引用”。

（3）如果函数返回值是一个对象，要考虑 return 语句的效率。例如

```
return String(s1 + s2);
```

这是临时对象的语法，表示“创建一个临时对象并返回它”。不要以为它与“先创建一个局部对象 temp 并返回它的结果”是等价的，如

```
String temp(s1 + s2);
return temp;
```

实质不然，上述代码将发生三件事。首先，temp 对象被创建，同时完成初始化；然后拷贝构造函数把 temp 拷贝到保存返回值的外部存储单元中；最后，temp 在函数结束时被销毁（调用析构函数）。然而“创建一个临时对象并返回它”的过程是不同的，编译器直接把临时对象创建并初始化在外部存储单元中，省去了拷贝和析构的化费，提高了效率。

类似地，我们不要将

```
return int(x + y);   // 创建一个临时变量并返回它
```

写成

```
int temp = x + y;
return temp;
```

由于内部数据类型如 int,float,double 的变量不存在构造函数与析构函数，虽然该“临时变量的语法”不会提高多少效率，但是程序更加简洁易读。

7.4 其它建议

【建议 1】函数的功能要单一，不要设计多用途的函数。

【建议 2】函数体的规模要小，尽量控制在 50 行代码之内。

【建议 3】尽量避免函数带有“记忆”功能。相同的输入应当产生相同的输出。

带有“记忆”功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某种“记忆状态”。这样的函数既不易理解又不利于测试和维护。在 C/C++语言中，函数的 static 局部变量是函数的“记忆”存储器。建议尽量少用 static 局部变量，除非必需。

【建议 4】不仅要检查输入参数的有效性，还要检查通过其它途径进入函数体内的变量的有效性，例如全局变量、文件句柄等。

【建议 5】用于出错处理的返回值一定要清楚，让使用者不容易忽视或误解错误情况。

8.参考文献

更多内容，请参阅以下书籍：

[1] 林锐_高质量 C++/C 编程指南(101 页)

[2] 华为编程规范(59 页)

[3] 嵌入式软件_C 语言编码规范(29 页)

[4] 王宜怀，陈建明，蒋银珍.基于 32 位 ColdFire 构建嵌入式系统

9.附录：C++/C 代码审查表

文件结构		
重要性	审查项	结论
	头文件和定义文件的名称是否合理？	
	头文件和定义文件的目录结构是否合理？	
	版权和版本声明是否完整？	
重要	头文件是否使用了 ifndef/define/endif 预处理块?	
	头文件中是否只存放“声明”而不存放“定义”	
	……	
程序的版式		
重要性	审查项	结论
	空行是否得体？	
	代码行内的空格是否得体？	
	长行拆分是否得体？	
	“{” 和 “}” 是否各占一行并且对齐于同一列？	
重要	一行代码是否只做一件事？如只定义一个变量，只写一条语句。	
重要	If、for、while、do 等语句自占一行，不论执行语句多少都要加“{}”。	
重要	在定义变量（或参数）时，是否将修饰符 * 和 & 紧靠变量名？	
	注释是否清晰并且必要？	
重要	注释是否有错误或者可能导致误解？	
重要	类结构的 public，protected，private 顺序是否在所有的程序中保持一致？	
	……	
命名规则		
重要性	审查项	结论
重要	命名规则是否与所采用的操作系统或开发工具的风格保持一致？	
	标识符是否直观且可以拼读？	
	标识符的长度应当符合“min-length &&	

	max-information”原则？	
重要	程序中是否出现相同的局部变量和全部变量？	
	类名、函数名、变量和参数、常量的书写格式是否遵循一定的规则？	
	静态变量、全局变量、类的成员变量是否加前缀？	
	……	
表达式与基本语句		
重要性	审查项	结论
重要	如果代码行中的运算符比较多，是否已经用括号清楚地确定表达式的操作顺序？	
	是否编写太复杂或者多用途的复合表达式？	
重要	是否将复合表达式与“真正的数学表达式”混淆？	
重要	是否用隐含错误的方式写 if 语句？例如 （1）将布尔变量直接与 TRUE、FALSE 或者 1、0 进行比较。 （2）将浮点变量用“==”或“！=”与任何数字比较。 （3）将指针变量用“==”或“！=”与 NULL 比较。	
	如果循环体内存在逻辑判断，并且循环次数很大，是否已经将逻辑判断移到循环体的外面？	
重要	Case 语句的结尾是否忘了加 break？	
重要	是否忘记写 switch 的 default 分支？	
重要	使用 goto 语句时是否留下隐患？例如跳过了某些对象的构造、变量的初始化、重要的计算等。	
	……	
常量		
重要性	审查项	结论
	是否使用含义直观的常量来表示那些将在程序中多次出现的数字或字符串？	
	在 C++ 程序中，是否用 const 常量取代宏常量？	
重要	如果某一常量与其它常量密切相关，是否在定义中包含了这种关系？	
	是否误解了类中的 const 数据成员？因为 const 数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的。	
	……	

函数设计		
重要性	审查项	结论
	参数的书写是否完整？不要贪图省事只写参数的类型而省略参数名字。	
	参数命名、顺序是否合理？	
	参数的个数是否太多？	
	是否使用类型和数目不确定的参数？	
	是否省略了函数返回值的类型？	
	函数名字与返回值类型在语义上是否冲突？	
重要	是否将正常值和错误标志混在一起返回？正常值应当用输出参数获得，而错误标志用 return 语句返回。	
重要	在函数体的“入口处”，是否用 assert 对参数的有效性进行检查？	
重要	使用滥用了 assert？ 例如混淆非法情况与错误情况，后者是必然存在的并且是一定要作出处理的。	
重要	return 语句是否返回指向“栈内存”的“指针”或者“引用”？	
	是否使用 const 提高函数的健壮性？const 可以强制保护函数的参数、返回值，甚至函数的定义体。“Use const whenever you need”	
	……	
其它常见问题		
重要性	审查项	结论
重要	数据类型问题： （1）变量的数据类型有错误吗？ （2）存在不同数据类型的赋值吗？ （3）存在不同数据类型的比较吗？	
重要	变量值问题： （1）变量的初始化或缺省值有错误吗？ （2）变量发生上溢或下溢吗？ （3）变量的精度够吗？	
重要	逻辑判断问题： （1）由于精度原因导致比较无效吗？ （2）表达式中的优先级有误吗？ （3）逻辑判断结果颠倒吗？	

重要	循环问题： （1）循环终止条件不正确吗？ （2）无法正常终止（死循环）吗？ （3）错误地修改循环变量吗？ （4）存在误差累积吗？	
重要	错误处理问题： （1）忘记进行错误处理吗？ （2）错误处理程序块一直没有机会被运行？ （3）错误处理程序块本身就有毛病吗？如报告的错误与实际错误不一致，处理方式不正确等等。 （4）错误处理程序块是“马后炮”吗？如在被它被调用之前软件已经出错。	
重要	文件 I/O 问题： （1）对不存在的或者错误的文件进行操作吗？ （2）文件以不正确的方式打开吗？ （3）文件结束判断不正确吗？ （4）没有正确地关闭文件吗？	