Ninebot 研发部 C 语言编程规范

版本/状态	修订人	参与者	起止日期	备注
V1.0	柴富华		2016-8-14	创建文档
V1.1 正式文件	任光伟		2019-5-23	根据 Ninebot 研发部开发环境进行修订
V1.2	柴富华		2019-6-7	修正部分内容，增加示例

Segway-Ninebot
地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add ／ 1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

目录

Segway-Ninebot
地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

Segway-Ninebot
地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add ／ 1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

第 1 章 文件结构

在 C 程序编程的源文件中通常分为两个文件。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。

1.1 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（参见示例 1-1），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。

```
/**
 * @file audio.c
 * @author Segway Insight
 * @brief 用于播放声音
 * @version 0.1
 * @date 2019-05-23
 *
 * @copyright Copyright (c) 2019 纳恩博（北京）科技有限公司
 * All rights reserved.
 *
 */
```

示例1-1 版权和版本的声明

【规则 1-1-1】EE 研发程序库发布版本要与源文件版本对应。

【规则 1-1-2】所有文件采用 GB2312 编码

【建议 1-1-1】产品线开发程序可忽略版本与日期。

批注：由于产品线开发程序部分几乎为最终端的程序，不会提供给外部或者其他部门使用，因此可以忽略版本日期，仅在一个指明工程信息的文件里保留版本信息即可。

1.2 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 1-1）。

（2）预处理块。

（3）函数和类结构声明等。

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com www.ninebot.com

假设头文件名称为 nb_lib.h，头文件的结构参见示例 1-2。

【规则 1-2-1】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 1-2-2】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 1-2-3】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）。
【规则 1-2-4】头文件中只存放“声明”而不存放“定义”。
【规则 1-2-5】禁止在头文件中出现象 extern int value 这类声明。

```
/**
 * @file nb_lib.h
 * @author your name (you@ninebot.com)
 * @brief
 * @version 0.1
 * @date 2019-05-26
 *
 * @copyright Copyright (c) 2019 纳恩博（北京）科技有限公司
 */

/* 防止重复引用 ----------------------------------------------------------------*/
#ifndef _NB_LIB_H_
#define _NB_LIB_H_
/* 头文件包含 ------------------------------------------------------------------*/
#include "crypto.h"
/* 类型定义 --------------------------------------------------------------------*/
typedef struct
{
    int simple;
}export_t;

/* 宏定义 ----------------------------------------------------------------------*/
#define LIB_DEF_DATA     123

/* 宏表达式 --------------------------------------------------------------------*/
#define LIB_MAX(a, b) (((a) > (b)) ? (a) : (b))

/* 函数声明 --------------------------------------------------------------------*/
int print_time(int hour, int min, int second)

#endif // _NB_LIB_H_
文件尾空行
```

示例 1-2 C++/C 头文件的结构

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com www.ninebot.com

1.3 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 1-1）。

（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 nb_lib.c，定义文件的结构参见示例 1-3。

```
/**
 * @file nb_lib.c
 * @author your name (you@ninebot.com)
 * @brief 简述文件作用
 * @version 0.1
 * @date 2019-05-26
 *
 * @copyright Copyright (c) 2019 纳恩博（北京）科技有限公司
 */

/* 头文件 ------------------------------------------------------------------*/
#include "nb_lib.h"
#include <stdio.h>

/* 类型定义 ----------------------------------------------------------------*/
typedef struct _foo
{
    int a;
    int b;
}foo_t;

typedef enum _bar
{
    BAR_A,
    BAR_B
}bar_t;

/* 宏定义 ------------------------------------------------------------------*/
#define LIB_MAX_NUM 100

/* 宏表达式 ----------------------------------------------------------------*/
```

```
#define LIB_MIN(a, b) (((a) < (b)) ? (a) : (b))

/* 静态变量 ------------------------------------------------------------------*/
static bar_t _bar = BAR_A;

/* 静态函数声明 --------------------------------------------------------------*/
static int _get_bar_state(bar_t *p_bar);

/* 外部函数声明 --------------------------------------------------------------*/
extern void get_a_value(int *a);

/* 全局变量 ----------------- ------------------------------------------------*/
int g_sc_abc;  //全局有符号字符
int g_uc_abc;  //全局无符号字符
int g_ss_abc;  //全局有符号短整型
int g_us_abc;  //全局无符号短整型
int g_si_abc;  //全局有符号整型
int g_ui_abc;  //全局无符号整型

int print_time(int hour, int min, int second)
{
    if(hour < 0 || min < 0 || second < 0)
    {
        return -1;
    }
    if(hour > 23 || min > 59 || second > 59)
    {
        return -2;
    }

    printf("%02d:%02d:%02d", hour, min, second);

    return 0;
}
函数间空行
static int _get_bar_state(bar_t *p_bar)
{
    *p_bar = _bar;
    return 0;
}
文件尾空行
```

示例 1-3 C++/C 定义文件的结构

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

1.4 头文件的作用

（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。

所以，.h 文件可以理解为.c 文件的“说明书”。一个规范的程序库，一般只需要看其.h 文件便能知道如何使用。

【规则 1-4-1】EE 研发程序库的头文件每个函数声明要有注释。

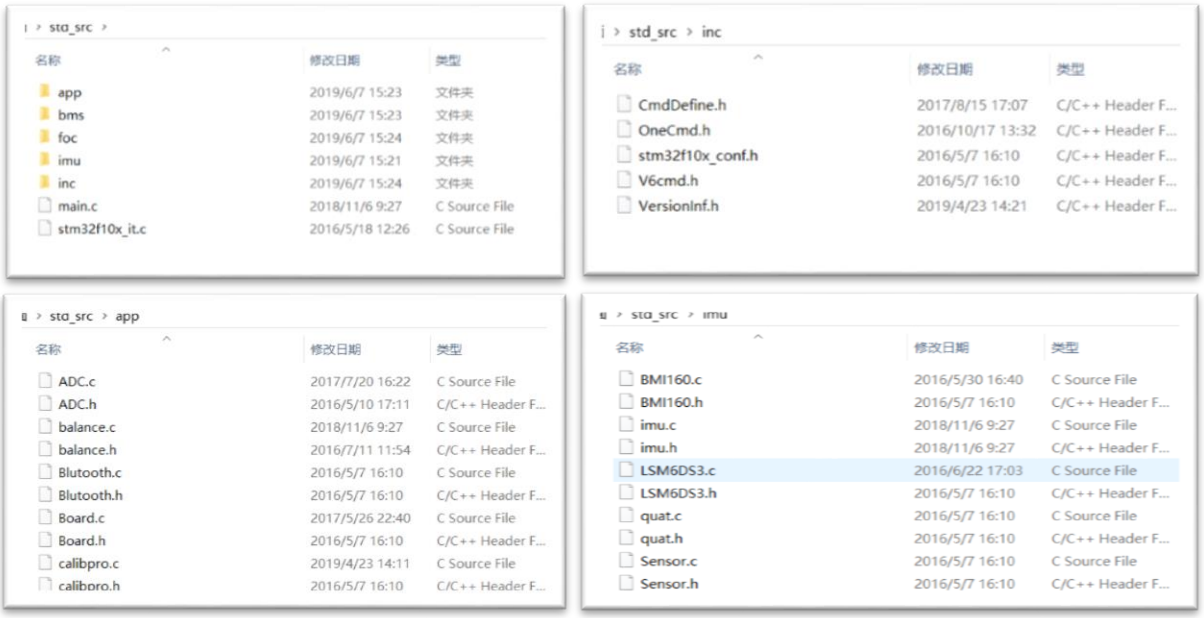
1.5 目录结构

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。

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。

【规则 1-5-1】相关联的 c 文件与 h 文件放在一个目录中。

【规则 1-5-2】c 文件与 h 文件尽量命名相同。



示例 1-4 C++/C 源文件目录结构

Segway-Ninebot
地址 / 北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel / +86-010-8482 8002）
Add / 1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

第 2 章 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。差的程序“书法”如螃蟹爬行，让人看得索然无味，更令维护者烦恼有加。请程序员们学习程序的“书法”，弥补大学计算机教育的漏洞，实在很有必要。

2.1 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 2-1-1】在每个类声明之后、每个函数定义结束之后都要加空行。参见示例 2-1（a）

【规则 2-1-2】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。参见示例 2-1（b）

```
函数间空行
void Function1(…)
{
  …
}
函数间空行
void Function2(…)
{
  …
}
函数间空行
void Function3(…)
{
  …
}
```

示例 2-1(a) 函数之间的空行

```
函数内空行
while (condition)
{
    statement1;
    函数内空行
    if (condition)
    {
        statement2;
    }
    else
    {
        statement3;
    }
    函数内空行
    statement4;
}
```

示例 2-1(b) 函数内部的空行

2.2 代码行

【规则 2-2-1】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。

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com www.ninebot.com

【规则 2-2-2】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。

示例 2-2（a）为风格良好的代码行　　示例 2-2（b）为风格不良的代码行。

建议：
`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();`
`}`
函数内空行
`other();`
	`for (initialization; condition; update)`
	`  dosomething();`
	`other();`

示例 2-2(a) **风格良好**的代码行　　示例 2-2(b) **风格不良**的代码行

【建议 2-2-1】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;     // 定义并初始化 width
int height = 10;    // 定义并初始化 height
int depth = 10;     // 定义并初始化 depth
int a[4] = {0};     // 定义并初始化数组
struct s abc = {0}; // 定义并初始化结构体
int *p = NULL;      // 定义并初始化指针
```

2.3 代码行内的空格

【规则 2-3-1】关键字之后要留空格。像 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。像 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 2-3-2】函数名之后不要留空格，紧跟左括号‘(’，以与关键字区别。

【规则 2-3-3】‘(’向后紧跟，‘)’、‘,’、‘;’向前紧跟，紧跟处不留空格。

【规则 2-3-4】‘,’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

Segway-Ninebot
地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

格，如 for (initialization; condition; update)。

【规则 2-3-5】赋值操作符、比较操作符、算术操作符、逻辑操作符、位与操作符，如“=”、“+=” “>=”、“<=”、“+”、“*”、“%”、“&&”、“||”、“<<”，“^”等二元操作符的前后应当加空格。

【规则 2-3-6】一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。

【规则 2-3-7】像“[]”、“.”、“->”这类操作符前后不加空格。

```
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

x = ((a) < (b)) ? (a) : (b);              // 良好的风格
x=a<b?a:b;                                // 不良的风格

int *x = &y;                              // 良好的风格
int * x = & y;                            // 不良的风格
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();
```

示例 2-3 代码行内的空格

2.4 对齐

【规则 2-4-1】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 2-4-2】{ }之内的代码块在‘{’右边数格处左对齐。

示例 2-4（a）为风格良好的对齐

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

示例 2-4（b）为风格不良的对齐。

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


Segway-Ninebot
地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

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：
{
…
{
…
}
…
}	

示例 2-4(a) **风格良好**的对齐　　示例 2-4(b) **风格不良**的对齐

2.5 长行拆分

【建议 2-5-1】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 2-5-2】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();
}
```

示例 2-5 长行的拆分

2.6 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，是个有争议的话题。

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 2-6-1】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
```

2.7 注释

程序块的注释符为“/*…*/”，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。参见示例 2-6。

题外话：很多程序员不喜欢写注释，主要是因为三个心理：一、我写的程序没人看；二、我写的程序没人看得懂；三、我写的程序我也不会写注释。在纳恩博你完全可以抛开前两个心理，如果你有第三个，请重写你的程序。

【规则 2-7-1】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。原则上要求注释要占到程序总体的 1/3。

【规则 2-7-2】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 2-7-3】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 2-7-4】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 2-7-5】尽量避免在注释中使用缩写，特别是不常用缩写。
【规则 2-7-6】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 2-7-8】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。
【规则 2-7-9】EE 研发内部要求使用 Doxygen 文档化注释，建议其他部门也使用文档化注释。

```
u8 bms_init_capacity_by_cell_voltage_and_current(void)
{
    s32 temp32 = 0;
    u8 i = 0;  //循环变量

    //根据电流修正电芯电压
    bq_get_current((s32*)&g_current);    //读取系统电流
    temp32 = g_cap_cell_vol;                    //用电芯电压初始化容量
    temp32 += g_current * m_v_cell_r / 1000;       //增加电流补偿电压
    //初始化滑模滤波器
    for(i = 0; i < HUAMO_BUF_LEN; i++)
    {
        m_bal_vol_huamo_buf[i] = temp32;
    }
    m_mAh_vol = bms_get_mAh_by_vol(temp32);     //根据电压来计算剩余的电量
    m_crt_cap = m_mAh_kulun = m_mAh_vol;     //将显示容量、库伦容量与电压容量统一
    set_bms_cmd_map_value(BMS_INF_CRT_CAP, m_mAh_vol); //将 BMS 的容量写入 Flash 的 MAP 表
    //计算 PACK 的电量百分比
    temp32 = m_mAh_Vol;
    temp32 = temp32 * 100 / g_full_cap;
    temp32 = temp32 > 100 ? 100 : (temp32 < 0 ? 0 : temp32);
    g_power_pct = temp32;
    //计 PACK 的电量百分比写入 Flash 的 MAP 表
    set_bms_cmd_map_value(BMS_INF_CAP_PCT, g_power_pct);
    return 1;
}
```

示例 2-6 程序的注释

```
/**
 * @brief 制作读取帧
 *
 * @param index 读取位置
 * @param read_len 要读取的字节数
 * @param msg 返回帧
 * @param msg_len 返回帧长度
 * @return int 错误码
 */
```

```
if (…)
{
   …
   while (…)
   {
      …
   } // end of while
   …
```


Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

	} // end of if

示例 2-7 程序的注释

Segway-Ninebot
地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add ／ 1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

第 3 章 命名规则

3.1 共性规则

本节论述的共性规则是被大多数程序员采纳的，我们应当在遵循这些共性规则的前提下，再扩充特定的规则，如 3.2 节。

【规则 3-1-1】标识符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：

好	差
current_value	now_value

【规则 3-1-2】标识符的长度应当符合“min-length && max-information”原则。

好

```
int price_count_reader;    // 无缩写
int num_errors;            // "num" 是一个常见的写法
int num_dns_connections;   // 人人都知道 "DNS" 是什么
```

差

```
int nerr;              // 含糊不清的缩写
int n_comp_conns;      // 含糊不清的缩写
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;          // 删减了若干字母.
```

【规则 3-1-3】命名规则尽量与所采用的操作系统或开发工具的风格保持一致。

例如 Windows 应用程序的标识符通常采用“大小写”混排的方式，如 AddChild。而单片机应用程序的标识符通常采用“小写加下划线”的方式，如 add_child。切忌别把这两类风格混在一起用。两种规则将会在后面详细讲解。

【规则 3-1-4】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);
```


Segway-Ninebot
地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

【规则 3-1-5】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。

【规则 3-1-6】变量的名字应当使用“名词”或者“形容词＋名词”。

例如：

```
float  value;
float  old_value;
float  new_value;
```

【规则 3-1-7】全局函数的名字应当使用“动词”或者“动词＋名词”（动宾词组）。结构体的成员函数应当只使用“动词”，被省略掉的名词就是对象本身。

例如：

```
draw_box();       // 全局函数
box->draw();      // 结构体的成员函数
```

【规则 3-1-8】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 min_value;
int max_value;

int set_value(…);
int get_value(…);
```

【建议 3-1-1】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 3-1-8】全局变量命名中包含作用域和类型

全局变量举例：g_sc_soooter

作用域	g_	全局变量	unsigned char g_uc_status;
	s_	静态变量	static unsigned int s_ui_systick;
类型	sc	有符号字符型	char sc_tempreture;
	uc	无符号字符型	unsigned char uc_percent;
	ss	有符号短整型	short ss_num;
	us	无符号短整型	unsigned short us_count;
	si	有符号整型	int ui_money;
	ui	无符号整型	unsigned int ui_high;
	f	单精度浮点型	float f_gyro;
	d	双精度浮点型	double d_length;
	p	指针	int *p_str;
	pv	void 指针	void *pv_str;

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com www.ninebot.com

3.2 简化“小驼峰”命名规则

【规则 3-2-1】类名和函数名用大写字母开头的单词组合而成。

例如：

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

【规则 3-2-2】变量和参数用小写字母开头的单词组合而成。

例如：

```
BOOL flag;
int  drawMode;
```

【规则 3-2-3】常量全用大写的字母，用下划线分割单词。

例如：

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

【规则 3-2-7】为了防止某一软件库中的一些标识符和其它软件库中的冲突，可以为各种标识符加上能反映软件性质的前缀。例如三维图形标准 OpenGL 的所有库函数均以 gl 开头，所有常量（或宏定义）均以 GL 开头。

例如：

```
#define NB_FRAME_LENGTH 100
unsigned char nb_crypto_type;
int nb_crypto_data(char *frame, int len);
```

3.3 单片机应用程序命名规则

【规则 3-3-1】函数名用小写字母和下划线组合而成。

例如：

```
void  draw(void);          // 函数名
void  set_value(int value);  // 函数名
```

【规则 3-3-2】变量和参数用小写字母开头的单词组合而成。

例如：

```
BOOL flag;
int  draw_mode;
```

【规则 3-3-3】常量及宏定义全用大写的字母，用下划线分割单词。

例如：

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

```
#define PRIV_DEFINE
```


Segway-Ninebot
地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

第 4 章 表达式和基本语句

有些人可能怀疑：连 if、for、while、goto、switch 这样简单的东西也要探讨编程风格，是不是小题大做？

其实很多程序员用隐含错误的方式写表达式和基本语句。

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

4.1 运算符的优先级

C++/C 语言的运算符有数十个，运算符的优先级与结合律如表 4-1 所示。注意一元运算符 + - * 的优先级高于对应的二元运算符。

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

表 4-1 运算符的优先级与结合律

优先级问题	表达式	经常误认为的结果	实际结果
.的优先级高于* ->操作符用于消除这个问题	`*p.f`	p 所指对象的字段 f `(*p).f`	对 p 取 f 偏移，作为指针，然后进行解除引用操作， `*(p.f)`
[]高于*	`int *ap[]`	ap 是个指向 int 数组的指针 `int (*ap)[]`	ap 是个元素为 int 指针的数组 `int *(ap[])`

函数（）高于*	int *fp()	fp 是个函数指针，所指函数返回 int int (*fp)()	fp 是个函数，返回 int * int * (fp())
== 和 ！= 高于位操作	(val & mask != 0)	（val & mask）!= 0	val & （mask != 0）
== 和!= 高于赋值操作	C = getchar() != EOF	(C = getchar()) != EOF	C = (getchar() != EOF)
算术运算符高于移位运算符	msb <<4 + lsb	(msb << 4) + lsb	Msb << (4+lsb)
逗号运算符在所有运算符中优先级最低	i = 1,2	i = (1,2)	(i = 1), 2

表 4-2 容易出错的优先级

【规则 4-1-1】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 4-1 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))
{
}
```

4.2 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 4-2-1】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 4-2-2】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 4-2-3】不要把程序中的复合表达式与“真正的数学表达式”混淆。

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com www.ninebot.com

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)  //结果为 a < b 判断结果再与 c 判断
```

4.3 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。

4.3.1 布尔变量与零值比较

【规则 4-3-1】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)
```

注：单片机编程中没有布尔型，因此不受该条约束

4.3.2 整型变量与零值比较

【规则 4-3-2】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)
```

4.3.3 浮点变量与零值比较

【规则 4-3-3】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

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

```
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）。

4.3.4 指针变量与零值比较

【规则 4-3-4】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)
```

4.3.5 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。

4.4 循环语句的效率

提高循环体效率的基本办法是降低循环体的复杂性。

【建议 4-4-1】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 4-4(b)的效率比示例 4-4(a)的高。

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

示例 4-4(a) 低效率：长循环在最外层

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

示例 4-4(b) 高效率：长循环在最内层

【建议 4-4-2】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。示例 4-4(c)的程序比示例 4-4(d)多执行了 N-1 次逻辑判断。并且由于前者总是进行逻辑判断，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 4-4(d)的写

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com www.ninebot.com

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 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();
}
```

表 4-4(c) 效率低但程序简洁

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

表 4-4(d) 效率高但程序不简洁

4.6 while 语句

while 循环会在循环开始前添加一条判断是否退出循环的语句，而 for 循环则在 while 的基础上，每次循环末尾都更新循环变量。需要注意的是区分两种中断循环的命令，一个是 continue，中断本次循环继续执行，另一个是 break，退出循环。

```
while(i < n)
{
  if (condition)
  {
      DoSomething();
      i++;
      continue;  //中断本次循环继续执行
  }
  else
  {
      DoOtherthing1();
      break;      //退出循环
  }
  DoOtherthing2();
  i++;
}
```

4.5 for 语句的循环控制变量

【规则 4-5-1】不可在 for 循环体内修改循环变量，防止 for 循环失去控制。

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

【建议 4-5-1】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 4-5(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 4-5(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 4-5(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++)
{
  …
}
```

示例 4-5(a) 循环变量属于半开半闭区间

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

示例 4-5(b) 循环变量属于闭区间

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;
    case value3:
    case value4:
        …
        break;
    default:
        …
        break;
}
```

【规则 4-6-1】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 4-6-2】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。

【建议 4-6-3】如果使用 switch 进行状态机切换，应尽可能配上 PlantUML 图

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

4.7 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。

【规则 4-7-1】只有当 goto 用于退出函数时，才可以使用。

```
void foo()
{
    ...
    if(exit_cond)
    {
        ...
        goto f_exit;
    }
    else
    {
        ...
    }
```


Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

```
    f_exit:
    //例如释放申请内存等
}
```


Segway-Ninebot
地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

第 5 章 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）。

5.1 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 5-1-1】 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
```

5.2 const 与 #define 的比较

C++ 语言可以用 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， 就是大括号要写好）。

（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 5-2-1】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。

5.3 常量定义规则

【规则 5-3-1】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，可以把耦合性高的不同模块的常量集中存放在一个公共的头文件中。

【规则 5-3-2】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 = 100.0;
const  float   DIAMETER = RADIUS * 2;
```


Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com www.ninebot.com

第 6 章 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，所以只是使函数的功能正确是不够的。本章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）。

6.1 参数的规则

【规则 6-1-1】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 set_value(int width, int height);  // 良好的风格
void set_value(int, int);               // 不良的风格
float get_value(void);                  // 良好的风格
float get_value();                      // 不良的风格
```

【规则 6-1-2】参数命名要恰当，顺序要合理。

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

```
void string_copy(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 dest 和 src。这样从名字上就可以看出应该把 src 拷贝到 dest。

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。

如将函数声明为：

```
void string_copy(char *dest, char *src);
```

这里如果不是标准的，很清楚能判断函数功能和参数的，最好加上类型等信息：

```
void string_copy(char *p_str_dest, char *p_str_src);
```

【规则 6-1-3】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 string_copy (char *dest, const char *src);
```

【建议 6-1-1】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，会影响调用性能。

6.2 返回值的规则

【规则 6-2-1】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。

【规则 6-2-2】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”人们一般预料不到！导致本例错误的责任并不在用户，是函数 getchar 误导了使用者。

【建议 6-2-1】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)； *（这里只是示例，显然 BOOL 无法包含正确，错误和文件结束等多于 2 个的状态）*

虽然 getchar 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 6-2-2】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 *dest, const char *src);
```

strcpy 函数将 src 拷贝至输出参数 dest 中，同时函数的返回值又是 dest。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”));
```


Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com www.ninebot.com

6.3 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 6-3-1】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。

```
void foo(int *a, SomeStruct *b)
{
#ifdef DBG
    assert(a);
    assert(b && b->type == 2);
#endif

    ...
    ...
}
```

【规则 6-3-2】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”。

6.4 其它建议

【建议 6-4-1】函数的功能要单一，不要设计多用途的函数。

【建议 6-4-2】函数体的规模要小，尽量控制在 50 行代码之内。

【建议 6-4-3】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

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com www.ninebot.com

器。建议尽量少用 static 局部变量，除非必需。

【建议 6-4-4】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 6-4-5】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。

6.5 使用断言

程序一般分为 Debug 版本和 Release 版本，Debug 版本用于内部调试，Release 版本发行给用户使用。

单片机的工程中，断言一般都是自定义的宏，其中包含打开开关，Debug 或者 Release 的配置都是由程序员控制的，因此你需要在 Debug 中定义相关的宏打开断言，在 Release 中关闭该宏。

断言 assert 是仅在 Debug 版本起作用的宏，它用于检查“不应该”发生的情况。示例 6-5 是一个内存复制函数。在运行过程中，如果 assert 的参数为假，那么程序就会中止（一般地还会出现提示对话，说明在什么地方引发了 assert）。

```
void  *memcpy(void *pvTo, const void *pvFrom, size_t size)
{
    assert((pvTo != NULL) && (pvFrom != NULL));     // 使用断言
    byte *pbTo = (byte *) pvTo;      // 防止改变 pvTo 的地址
    byte *pbFrom = (byte *) pvFrom;  // 防止改变 pvFrom 的地址
    while(size -- > 0 )
        *pbTo ++ = *pbFrom ++ ;
    return pvTo;
}
```

示例 6-5 复制不重叠的内存块

如果程序在 assert 处终止了，并不是说含有该 assert 的函数有错误，而是调用者出了差错，assert 可以帮助我们找到发生错误的原因。

```
#ifdef   USE_FULL_ASSERT   //使能断言的宏定义
#define assert_param(expr) ((expr) ? (void)0U : assert_failed((uint8_t *)__FILE__, __LINE__))
void assert_failed(uint8_t *file, uint32_t line);
#else
#define assert_param(expr) ((void)0U)  //如果没有使能，则断言函数为空
#endif /* USE_FULL_ASSERT */
```

【规则 6-5-1】使用断言捕捉不应该发生的非法情况。不要混淆非法情况与错误情况之间的区别，后者是必然存在的并且是一定要作出处理的。

【规则 6-5-2】在函数的入口处，使用断言检查参数的有效性（合法性）。

【建议 6-5-1】在编写函数时，要进行反复的考查，并且自问：“我打算做哪些假定？”一旦确定了的假定，就要使用断言对假定进行检查。

【建议 6-5-2】一般教科书都鼓励程序员们进行防错设计，但要记住这种编程风格可能会隐瞒错误。当进行

防错设计时，如果“不可能发生”的事情的确发生了，则要使用断言进行报警。

Segway-Ninebot
地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

第 7 章 其它编程经验

7.1 使用 const 提高函数的健壮性

看到 const 关键字，C++程序员首先想到的可能是 const 常量。这可不是良好的条件反射。如果只知道用 const 定义常量，那么相当于把火药仅用于制作鞭炮。const 更大的魅力是它可以修饰函数的参数、返回值，甚至函数的定义体。

const 是 constant 的缩写，“恒定不变”的意思。被 const 修饰的东西都受到强制保护，可以预防意外的变动，能提高程序的健壮性。所以很多 C++程序设计书籍建议：“Use const whenever you need”。

7.1.1 用 const 修饰函数的参数

如果参数作输出用，不论它是什么数据类型，也不论它采用“指针传递”还是“引用传递”，都不能加 const 修饰，否则该参数将失去输出功能。

const 只能修饰输入参数：

（1）如果输入参数采用“指针传递”，那么加 const 修饰可以防止意外地改动该指针，起到保护作用。

例如 string_copy 函数：

```
void string_copy(char *dest, const char *src);
```

其中 src 是输入参数，dest 是输出参数。给 src 加上 const 修饰后，如果函数体内的语句试图改动 src 的内容，编译器将指出错误。

（2）如果输入参数采用“值传递”，由于函数将自动产生临时变量用于复制该参数，该输入参数本来就无需保护，所以不要加 const 修饰。

例如不要将函数 void Func1(int x) 写成 void Func1(const int x)。

同理不要将函数 void Func2(A a) 写成 void Func2(const A a)。其中 A 为用户自定义的数据类型。

7.1.2 用 const 修饰函数的返回值

（1）如果给以“指针传递”方式的函数返回值加 const 修饰，那么函数返回值（即指针）的内容不能被修改，该返回值只能被赋给加 const 修饰的同类型指针。

例如函数

```
const char * get_string(void);
```

如下语句将出现编译错误：

```
char *str = get_string();
```

正确的用法是

```
const char *str = get_string();
```

（2）如果函数返回值采用“值传递方式”，由于函数会把返回值复制到外部临时的存储单元中，加 const 修饰没有任何价值。

例如不要把函数 int get_int(void) 写成 const int get_int(void)。

同理不要把函数 A GetA(void) 写成 const A GetA(void)，其中 A 为用户自定义的数据类型。

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com www.ninebot.com

7.2 提高程序的效率

程序的时间效率是指运行速度，空间效率是指程序占用内存或者外存的状况。

全局效率是指站在整个系统的角度上考虑的效率，局部效率是指站在模块或函数角度上考虑的效率。

【规则 7-2-1】不要一味地追求程序的效率，应当在满足正确性、可靠性、健壮性、可读性等质量因素的前提下，设法提高程序的效率。

【规则 7-2-2】以提高程序的全局效率为主，提高局部效率为辅。

【规则 7-2-3】在优化程序的效率时，应当先找出限制效率的“瓶颈”，不要在无关紧要之处优化。

【规则 7-2-4】先优化数据结构和算法，再优化执行代码。

【规则 7-2-5】有时候时间效率和空间效率可能对立，此时应当分析那个更重要，作出适当的折衷。例如多花费一些内存来提高性能。

【规则 7-2-6】不要追求紧凑的代码，因为紧凑的代码并不能产生高效的机器码。

7.3 一些有益的建议

【建议 7-3-1】当心那些视觉上不易分辨的操作符发生书写错误。

我们经常会把“＝＝”误写成“＝”，像“||”、“&&”、“<=”、“>=”这类符号也很容易发生“丢 1”失误。然而编译器却不一定能自动指出这类错误。

【建议 7-3-2】当心变量的初值、缺省值错误，或者精度不够。

【建议 7-3-3】当心数据类型转换发生错误。尽量使用显式的数据类型转换（让人们知道发生了什么事），避免让编译器轻悄悄地进行隐式的数据类型转换。

例如 double d_val = 1.2;

```
int si_val = 5;
double d_val2 = 0;
d_val2 = d_val + si_val; //此时 si_val 被隐式转换为 double 型
d_val2 = 5 / 2;          //此时 5 与 2 都是 int 型变量，没有隐式转换，因此 d_val2 结果为 2.000000
```

表达式中数值会向高精度转换，同精度不会转换。

d_val2 = 5 / 2 应显示转换其中任何一个数值为 double 型，则另外一个自动隐式转换为 double 型。

例如:

d_val2 = 5 / 2.0; 或 d_val2 = 5 / (double) 2;

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com www.ninebot.com

【建议 7-3-4】当心变量发生上溢或下溢，数组的下标越界。

```
//如下程序将 rbuff 内容解析到 app 中，在进行数组操作时，要时刻注意数组长度。
#define MAX_GPS_ITEM 19
static char app[MAX_GPS_ITEM][MAX_GPS_ITEM];
int gps_data_poll(gps_data_t *data)
{
    char rbuff[MAX_GPS_READ] = {0};
    int rl = 0;
    rl = uart_read_gps(gps_huart, (uint8_t *)rbuff, MAX_GPS_READ, 200);
    // do something
    for (int i = 0, j = 0, k = 0; (rbuff[i] != '\n') && (i < MAX_GPS_READ) && (j < MAX_GPS_ITEM) && (k
< MAX_GPS_ITEM); i++)       //注意此处要判断 rbuff 的长度与 app 数组的长度
    {
        if (rbuff[i] == '*')
        {
            break;
        }
        if (rbuff[i] == ',')
        {
            app[j][k] = '\0';
            j++;
            k = 0;
        }
        else
        {
            app[j][k] = rbuff[i];
            k++;
        }
    }
    // do something
}
```

【建议 7-3-5】尽量使用标准库函数，不要“发明”已经存在的库函数。

【建议 7-3-6】把编译器的选择项设置为最严格状态。

【建议 7-3-7】如果可能的话，使用 PC-Lint、LogiScope 等工具进行代码审查。

Segway-Ninebot
地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

附录 A ： 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		

Segway-Ninebot 地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory),66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com

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”	
	……	

Segway-Ninebot
地址／北京市海淀区西小口路66号中关村东升科技园北领地A-1号楼一层 （Tel／+86-010-8482 8002）
Add／1F A1 BLDG., ZGC DongSheng Science Park (Northern Territory), 66 Xixiaokou Rd. Haidian Disct. Beijing, China
www.segway.com
www.ninebot.com