C/C++编码规范

目录

[**1** **引言** 6](#_Toc10208122)

[**2** **排版** 6](#_Toc10208123)

[**3** **注释** 10](#_Toc10208133)

[**4** **命名规则** 13](#_Toc10208144)

[**5** **可读性** 14](#_Toc10208150)

[**6** **数据类型** 15](#_Toc10208154)

[**7** **常量** 15](#_Toc10208156)

[**8** **变量、结构** 17](#_Toc10208159)

[**9** **类** 18](#_Toc10208162)

[**10** **宏** 19](#_Toc10208167)

[**11** **函数、过程** 20](#_Toc10208169)

[**12** **错误代码** 20](#_Toc10208171)

[**13** **调试** 20](#_Toc10208173)

[**14** **内存管理** 21](#_Toc10208176)

[**15** **实践建议** 21](#_Toc10208178)

1. **引言**

为了提高源程序的质量和可维护性，最终提高公司软件产品生产力，我们有必要对公司软件产品的源程序的编写风格作出统一的规范约束。

本规范总则的内容包括：排版、注释、标识符命名、变量使用、代码可测性、程序效率、质量保证等。本规范采用以下的术语描述：

* 规则：编程时**强制必须遵守**的原则。
* 说明：对此规则或建议进行必要的解释。
* 示例：对此规则或建议从正、反两个方面给出例子。

1. **排版**
   1. **程序块要采用缩进风格编写，缩进允许4个空白字符或者一个TAB键，同一个模块的风格必须一致。(建议)**

建议使用4个空白字符，如使用TAB键，则应该在环境中也设置为4个空白。

* 1. **函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格，case语句下的情况处理语句也要遵从语句缩进要求。(建议)**
  2. **注释与所描述内容进行同样的缩排。(建议)**

说明：可使程序排版整齐，并方便注释的阅读与理解。

错误示例：

void example\_fun()

{

/\* code one comments \*/

CodeBlock One

/\* code two comments \*/

CodeBlock Two

}

应改为如下布局。

void example\_fun()

{

/\* code one comments \*/

CodeBlock One

/\* code two comments \*/

CodeBlock Two

}

* 1. **相对独立的程序块之间、变量说明之后必须加空行。(建议)**

错误示例：

if (!valid\_ni(ni))

{

// program code

}

repssn\_ind = ssn\_data[index].repssn\_index;

repssn\_ni = ssn\_data[index].ni;

正确示例：

if (!valid\_ni(ni))

{

// program code

}

repssn\_ind = ssn\_data[index].repssn\_index;

repssn\_ni = ssn\_data[index].ni;

* 1. **较长的语句要分成多行书写，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读，下一行至少有一个缩进。(建议)**

正确示例：

perm\_count\_msg.head.len = NO7\_TO\_STAT\_PERM\_COUNT\_LEN

+ STAT\_SIZE\_PER\_FRAM \* sizeof( \_UL );

act\_task\_table[frame\_id \* STAT\_TASK\_CHECK\_NUMBER + index].occupied

= stat\_poi[index].occupied;

act\_task\_table[taskno].duration\_true\_or\_false

= SYS\_get\_sccp\_statistic\_state( stat\_item );

report\_or\_not\_flag = ((taskno < MAX\_ACT\_TASK\_NUMBER)

&& (n7stat\_stat\_item\_valid (stat\_item))

&& (act\_task\_table[taskno].result\_data != 0));

* 1. **若函数或过程中的参数较长，则要进行适当的划分，按逗号分行。(强制)**

正确示例：

n7stat\_str\_compare((BYTE \*) & stat\_object,

(BYTE \*) & (act\_task\_table[taskno].stat\_object),

sizeof (\_STAT\_OBJECT));

* 1. **不允许把多个短语句写在一行中，即一行只写一条语句。(强制)**

错误示例：

rect.length = 0; rect.width = 0;

正确示例：

rect.length = 0;

rect.width = 0;

* 1. **程序块的分界符（如C/C++语言的大括号‘{’和‘}’），允许有两种方式，独占一行并且位于同一列，同时与引用它们的语句左对齐；或者放在行的结尾。在函数体的开始、类的定义、结构的定义、枚举的定义以及if、for、do、while、switch、case语句中的程序都要采用如上的缩进方式。(强制)**

正确示例：

for (...)

{

// program code

}// ~end for (...)

正确示例：

for (...) {

... // program code

}

* 1. **在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如－>），后不应加空格。(建议)**

说明：采用这种松散方式编写代码的目的是使代码更加清晰。由于留空格所产生的清晰性是相对的，所以，在已经非常清晰的语句中没有必要再留空格，如果语句已足够清晰则括号内侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在C/C++语言中括号已经是最清晰的标志了。

在长语句中，如果需要加的空格非常多，那么应该保持整体清晰，而在局部不加空格。给操作符留空格时不要连续留两个以上空格。

正确示例：

(1) 逗号、分号只在后面加空格。

int a, b, c;

(2)比较操作符, 赋值操作符"="、 "+="，算术操作符"+"、"%"，逻辑操作符"&&"、"&"，位域操作符"<<"、"^"等双目操作符的前后加空格。

if (current\_time >= MAX\_TIME\_VALUE)

a = b + c;

a \*= 2;

a = b ^ 2;

(3)"!"、"~"、"++"、"--"、"&"（地址运算符）等单目操作符前后不加空格。

\*p = 'a'; // 内容操作"\*"与内容之间

flag = !isEmpty; // 非操作"!"与内容之间

p = &mem; // 地址操作"&" 与内容之间

i++; // "++","--"与内容之间

(4)"->"、"."前后不加空格。

p->id = pid; // "->"指针前后不加空格

(5) if、for、while、switch等与后面的括号间应加空格，使if等关键字更为突出、明显。

if (a >= b && c > d)

1. **注释**
   1. **说明性文件（如头文件.h文件、.inc文件、.def文件、配置说明文件.cfg等）头部应进行注释，注释必须列出：版权说明、作者和功能简要说明。(强制)**

/\*

\* @(#)Blah.java 1.82 99/03/18

\*

\* Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.

\*

\* FIPS pub 180-1: Secure Hash Algorithm (SHA-1)

\* based on: http://csrc.nist.gov/fips/fip180-1.txt

\* implemented by Jun-ichiro itojun Itoh <itojun@itojun.org>

\*

\* @version 1.82 18 Mar 1999

\* @author Firstname Lastname

\*

\*/

* 1. **函数头部应进行注释，列出：函数的目的/功能、输入参数、输出参数、返回值、调用关系（函数、表）等。(强制)**

/\*\*

\* Description :checkout 提款

\* @param Hashtable cart info

\* @param OrderBean order info

\* @return result

\* @exception

\*/

int checkout(Hashtable \*htCart, OrderBean \*orderBean)

* 1. **不再有用的注释要删除，以保证注释与代码的一致性。(强制)**
  2. **避免在注释中使用缩写，特别是非常用缩写。(建议)**

说明：在使用缩写时或之前，应对缩写进行必要的说明。

* 1. **注释应与其描述的代码相近，对代码的注释应放在其上方或右方（对单条语句的注释）相邻位置，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。(建议)**

错误示例：

例1：

/\* International roamer calculate toll fee \*/

strcpy(deb.chargecode,CHARGE\_IDD);

strcpy(deb.calltype,CALLTYPE\_IDD);

例2：

strcpy(deb.chargecode,CHARGE\_IDD);

strcpy(deb.calltype,CALLTYPE\_IDD);

/\* International roamer calculate toll fee \*/

正确示例：

/\* International roamer calculate toll fee \*/

strcpy(deb.chargecode,CHARGE\_IDD);

strcpy(deb.calltype,CALLTYPE\_IDD);

* 1. **对于所有有特定含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其特定含义。变量、常量、宏的注释应放在其上方相邻位置或右方。(建议)**

正确示例：

/\* active statistic task number \*/

#define MAX\_ACT\_TASK\_NUMBER 1000

#define MAX\_ACT\_TASK\_NUMBER 1000 /\* active statistic task number \*/

* 1. **数据结构声明(包括数组、结构、类、枚举等)，如果其命名不是充分自注释的，必须加以注释。对数据结构的注释应放在其上方相邻位置，不可放在下面；对结构中的每个域的注释放在此域的右方。(强制)**

正确示例：

/\* structure definition for parameter request item\*/

typedef struct {

char \*name; /\* parameter name \*/

int type; /\* data type \*/

void \*value; /\* variable to store parameter value \*/

int need; /\* needful \*/

int size; /\* variable size \*/

int \*number; /\* variable to store parameter count \*/

int nummax; /\* arrary size \*/

} Para, \*pPara;

* 1. **在程序块的结束行右方加注释标记，以表明某程序块的结束。(建议)**

说明：当代码段较长，特别是多重嵌套时，这样做可以使代码更清晰，更便于阅读。

正确示例：

if (...)

{

// program code

while (index < MAX\_INDEX)

{

// program code

} // ~end while (index < MAX\_INDEX)

} // ~end if (...)

* 1. **命名中若使用特殊约定或缩写，则要有注释说明。(强制)**

说明：应该在源文件的开始之处，对文件中所使用的缩写或约定，特别是特殊的缩写，进行必要的注释说明。

* 1. **无用的代码不能注释掉。 （强制）**

要么删除， 要么使用条件编译屏蔽。

可以这样写：

#ifdef 0

/\* 无用的代码 \*/

#endif

1. **命名规则**
   1. **常量名称统一使用大写，标识符必须指示常量具体含义。(强制)**

#define MAX\_PATH 256

* 1. **变量名称禁止取单个字符（如i、j、k...），标识符必须指示常量具体含义，但是i、j、k作局部循环变量是允许的。(强制)**
  2. **类的方法的命名和过程、函数的命名一样。类的属性名应为名词。属性的命名规则和其他的变量相同。(建议)**
  3. **设置输入参数值的例程名应当以set为其前缀，获取数值的例程名应当以get为其前缀。(建议)**
  4. **标识符的命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。(建议)**

说明：较短的单词可通过去掉“元音”形成缩写；较长的单词可取单词的头几个字母形成缩写；一些单词有大家公认的缩写。

1. **可读性**
   1. **注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。(强制)**

说明：防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

正确示例：

word = (high << 8) | low (1)

if ((a | b) && (a & c)) (2)

if ((a | b) < (c & d)) (3)

错误示例：

high << 8 | low

a | b && a & c

a | b < c & d

由于

high << 8 | low = ( high << 8) | low,

a | b && a & c = (a | b) && (a & c)，

(1)(2)不会出错，但语句不易理解；

a | b < c & d = a | （b < c） & d，(3)造成了判断条件出错。

* 1. **源程序中关系较为紧密的代码应尽可能相邻。(强制)**

说明：便于程序阅读和查找。

错误示例：

rect.length = 10;

char\_poi = str;

rect.width = 5;

正确示例：

rect.length = 10;

rect.width = 5; // 矩形的长与宽关系较密切，放在一起。

char\_poi = str;

* 1. **不要使用难懂的技巧性很高的语句，除非很有必要时。(强制)**

说明：高技巧语句不等于高效率的程序，实际上程序的效率关键在于算法。

错误示例：

\* stat\_poi ++ += 1;

\* ++ stat\_poi += 1;

正确示例：

\*stat\_poi += 1;

stat\_poi++; // 此二语句功能相当于“ \* stat\_poi ++ += 1; ”

++ stat\_poi;

\*stat\_poi += 1; // 此二语句功能相当于“ \* ++ stat\_poi += 1; ”

1. **数据类型**

**5-1 禁止数据类型隐性地从高精度向低精度转换 (强制)**

例如int转换为short型或者char型。

double型转换为float型。

1. **常量**
   1. **C或者C++文件中，除允许出现-1，0，1三个数字外，其他数字都必须以常量方式定义，同一类常量应该统一格式，当-1, 0, 1有非常具体的意义时，也应该以常量代替。（强制）**

所有有特殊意义的数字都应该使用使用常量代替。

魔幻数字有百害而无一利，应该完全杜绝使用，特殊情况下可以使用数字。例如strncpy(dest, src, 5);

数字0主要可能出现在循环起始、数据结构初始化等地方。

-1主要出现在与函数结果比较的地方。

1应该是可以不需要出现的，不过有的地方确实以1为下标的。

给数组连续赋值和从数组中连续读取值可以采用以下方法实现：

int index=0;

a = array[index++];

b = array[index++];

c = array[index++];

这种方式比以下方式：

a = array[0];

b = array[1];

c = array[2];

好一些，好处如下：

1. 如果下标需要修改为从1开始，那么只需要修改index = 1即可，而第二种方法需要逐个修改。
2. 程序更加整齐，眼睛瞟一下就能够发现异常情况，特别是在项数比较多的情况下，只需要检查左边即可，右边都是一样的。第二种情况需要检查左右两边，容易出错。
3. 如果数据内容顺序做了修改，只需要挪动相应的程序行即可，第二种方法必须修改索引。

* 1. **避免直接使用字符串，大部分地方可以用常量代替，有些地方可以以整数代替（即字符串的索引）。(建议)**

当前公司的业务主要是处理数值，字符串主要用做提示和说明信息，对计算机来说，处理整数的效率比处理字符串的效率不止高2个数量级。字符串中涉及到空白，大小写，编码等情况，容易出错。

例如需要显示无效的参数，不直接使用 “invalid param”

而是使用常数：

#define INVALID\_PARAM “invalid param”

或者定义数组： static char errmsg[]={“invalid param”};

#define INVALID\_PARAM 0

使用字符串 “20080102083445”表示时间，可以改为用time\_t或者其他时间类或者结构来表示。

1. **变量、结构**
   1. **绝大部分情况下禁止使用系统级的全局变量，而文件级的静态只读的全局变量则是可以使用的，函数库中不能输出全局变量。(强制)**

全局变量在多线程应用中可能带来很多问题，推荐使用其他开发软件中的方法。定义一个结构(在C++中可以定义一个这样的类)

typedef struct \_context

{

/\* 需要用作全局变量内容放在这里 \*/

}CONTEXT;

系统中绝大部分函数都把这个参数(CONTEXT \*ctx)作为第一个参数，这样就没有全局变量的必要了。

特殊情况的例子如下：

系统中的所有错误信息可以通过只读的全局变量集中存放，放在一个c/cpp文件中，以static标识。

例如：static char errmsg[]={“error1”,

“error2”};

这种做法是无害的。

这种方式通常用来保存错误信息或者提示信息。

* 1. **结构的定义(强制)**

结构的定义要求使用unix中的定义方式，全部以小写表示。例如：

struct timeval {

long tv\_sec; /\* seconds since Jan. 1, 1970 \*/

long tv\_usec; /\* /\* and microseconds \*/

};

结构的定义方式为强制，结构中成员变量的定义方式建议使用结构名的缩写，加上下划线再加变量。

1. **类**
   1. **类名定义(建议)**

类的名字必须由一个或数个能表达该类的意思的大写字母开头而其它字母都小写的单词或缩写组成。这样能使这个类的名称能更容易被理解，

例如：

class Students

{

public:

int getName(char \*buf, int \*len);

int setName(char \*buf, int len);

private:

char name[MAX\_NAME\_LEN];

};

* 1. **成员变量(强制)**

变量名由一个或数个能表达该变量的意思的单词组成，使用大小写混合的方式，每一个词是小写，后面的词第一个字母大写，其它小写。

* 1. **成员函数(强制)**

成员函数的名称应该是一个动词，使用大小写混合的方式，每一个词是小写，后面的词第一个字母大写，其它小写。

* 1. **禁止在C++类中使用非基本类型的静态成员变量。(强制)**

如果在类中使用其他类的实例作为静态变量，系统在连接和初始化时可能会出现问题，可以使用静态非基本类型的成员变量指针。

非基本类型的指针仍然是基本类型。

1. **宏**
   1. **宏定义(建议)**

宏定义应该全部使用大写字母，用下划线隔开多个词汇，当宏中语句比较多时，推荐使用以下方法定义，例如：

#define PERF\_CHECK(x) do{ if((x)==0)\

Xxxx; \

else \

Xxxx;\

}while(0)

即用do{} while(0)，把语句包起来，linux的源代码中基本上都是这样做的，这是防止宏扩展时出问题的最好办法，而且不会增加运行时CPU的负担，因为do{}while(0)语句会被编译器优化掉。

1. **函数、过程**
   1. **函数的规模尽量限制在200行以内。(建议)**

说明：不包括注释和空格行。

1. **错误代码**
   1. **错误代码的定义。(强制)**

应该用数值来表示错误代码，其格式应该一样，例如用ERR作为前缀，后面用动宾结构说明原因，同类的错误应该合并，例如：

#define ERR\_BASE 1000

#define ERR\_OPEN\_FILE (ERR\_BASE+1)

#define ERR\_CREATE\_FILE (ERR\_BASE+2)

错误内容可以通过一个函数，例如 strerror返回。

公司将来可以收集所有的错误代码和文字信息，统一命名并应用到所有项目中。

1. **调试**
   1. **以DEBUG宏是否定义产生调试版本和发布版本。(强制)**

DEBUG宏是一个总开关，可以在DEBUG宏的基础上定义其他宏。如果DEBUG宏没有定义，那么其他宏都是没有定义的。即使DEBUG宏定义了，可能还必须定义其他宏来调试不同的内容。例如定义DEBUG\_DB来表示调试数据库的相关内容。

* 1. **使用VERIFY宏。(建议)**

#ifdef DEBUG

#define VERIFY(e) \

(!(e) ? (void)0 : \

(fprintf(stderr, "Verify failure: %s:%d: \"%s\"", \

\_\_FILE\_\_, \_\_LINE\_\_, #e), abort()))

#else

#define VERIFY(e) (e)

#endif

如果定义了DEBUG宏，这个宏有效。否则直接执行表达式e。

这个宏当e的值为非0时，将打印错误行和内容，并且退出。

如果遵循了函数的返回规范，把这个宏加到函数上，可以保证对每个函数的返回值都进行检查。

1. **内存管理**
   1. **内存管理应该遵循谁申请谁释放、谁提供申请方法谁就提供释放方法的原则。(建议)**

如果一个类或者模块提供了一个函数，内部包含了申请内存的功能。应该同时提供一个函数，允许释放这些内存。而不应该让调用程序通过free或者delete来释放内存。这样有如下好处：

(1).该类或者模块内部可以改变内部申请和释放内存的方式，只需要申请和释放方式互相对应即可。

(2).外部调用程序不用关心类或者模块的内部实现，只需配对使用接口即可。

(3).出现内存泄漏时可以分别测试定位，降低了模块之间的藕合。

1. **实践建议**
   1. **应用中不直接使用C/C++语言中的数据类型**

应用中不直接使用int、void \*、char \*类型除外的C/C++的数据类型。

int类型的使用主要用作索引、次数等，因为索引、次数都是从0开始，基本上不会使整数溢出。

不应该使用C++的bool类型，直接用整数代替，因为整数具有bool所有的功能，而且将来可以附加其他信息。

不应该使用void类型，void只能做函数返回值，但void作返回值时无法带回任何内容，浪费了从函数内部带回值的机会，可以用int作为函数结果，然后返回0即可，这样调用函数可以正常判断函数是否成功，与原来没有什么不同，将来如果函数实现需要修改，需要返回值的时候，只需要修改函数内部即可，无需修改调用部分，减少函数内部与调用部分的藕合，void\*是可以使用的。

一个项目中，其使用的数据类型应该通过typedef统一定义使用，可以根据unix的使用惯例定义，例如：pid\_t，pthread\_t，key\_t，time\_t，ssize\_t，size\_t，regex\_t，regmatch\_t等。

公司内部的变量可以使用money\_t，cumulate\_t等数据类型。

这样做的好处很多，例如：

(1).发现与money\_t相关的内容要修改时，容易迅速定位，而且不会漏掉；

(2).不同时期的模块同时使用时(某些为重用模块)，若两个模块的money\_t的数据类型不一致时，容易统一；

(3).若在设计时给money\_t预计的数据类型不符合要求时，只需要重新定义即可。例如要把money\_t由double型改为long long型都是很方便的，如果有编译不过的地方，也容易迅速定位，如果不使用money\_t直接使用double，要改为long long型的话就要仔细查找哪些double要改，哪些不要改。

(4).维护和测试人员容易理解程序。

公司可以先要求项目内部使用指定的定义，等执行到一定程度后，可以统一定义一些数据类型，要求所有项目使用。

* 1. **协议中的数据类型必须指定长度**

与外部设备通讯的协议禁止直接使用任何C/C++的数据类型，也禁止使用money\_t等数据类型，必须定义成INT8，UINT8，FLOAT64，INT64等说明长度的数据类型。这样可以保证协议的可移植性和可重用性。

* 1. **C语言中函数的定义**

函数定义方式建议学习unix中多线程函数定义的方法。pthread\_attr\_getstackaddr

pthread\_attr\_getdetachstate

pthread\_attr\_init

等，即把函数名称按词汇用下划线分开。

如果把C函数和C++的成员函数类比，那么相当于把C++的类摊平，即pthread\_attr相当于C++的类。

* 1. **C语言中函数的返回**

函数的返回，应该遵照unix的惯例，所有的函数都使用整数类型返回，0表示成功，大于0表示错误码。如果需要返回多个值，可以通过参数返回。

Unix系统原来使用errno表示错误码，函数返回-1表示失败。但errno是全局变量，对于多线程应用来说，无法使用这种方法，所以Unix后来定义的函数做了修改。

可以参照Unix的中线程函数：

int pthread\_attr\_getstacksize(const pthread\_attr\_t \*attr, size\_t \*stacksize)

值通过后面的参数返回。