**编程规范讨论稿-中科虹霸**

**2012-5-7**

**主要是参考了华为的编程规范，还有一些是GOOGLE的。**

软件编程规范总则

**一、 排版**

1、程序块要采用缩进风格编写，缩进的空格数为4个。

说明：对于由开发工具自动生成的代码可以有不一致。

2、相对独立的程序块之间、变量说明之后必须加空行。

示例：如下例子不符合规范。

if (!valid\_ni(ni))

{

... // program code

}

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

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

应如下书写

if (!valid\_ni(ni))

{

... // program code

}

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

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

3、较长的语句（>80 字符或横向一屏内）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。

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));

4、循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分，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。

示例：

if ((taskno < max\_act\_task\_number)

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

{

... // program code

}

for (i = 0, j = 0; (i < BufferKeyword[word\_index].word\_length)

&& (j < NewKeyword.word\_length); i++, j++)

{

... // program code

}

for (i = 0, j = 0;

(i < first\_word\_length) && (j < second\_word\_length);

i++, j++)

{

... // program code

}

5、若函数或过程中的参数较长，则要进行适当的划分。 **超过四个以上建议使用结构体进行封装。**

示例：

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

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

sizeof (\_STAT\_OBJECT));

n7stat\_flash\_act\_duration( stat\_item, frame\_id \*STAT\_TASK\_CHECK\_NUMBER

+ index, stat\_object );

6、不允许把多个短语句写在一行中，即一行只写一条语句。

示例：如下例子不符合规范。

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

应如下书写

rect.length = 0;

rect.width = 0;

7、if、for 、do、while 、case、switch 、default等语句自占一行，且if、for 、

do、while等语句的执行语句部分无论多少都要加括号{}。

示例：如下例子不符合规范。

if (pUserCR == NULL) return;

应如下书写：

if (pUserCR == NULL)

{

return;

}

8、对齐只使用空格键，不使用TAB 键。

说明：以免用不同的编辑器阅读程序时，因TAB 键所设置的空格数目不同而造成程序布局

不整齐，不要使用BC作为编辑器合版本，因为 BC会自动将 8 个空格变为一个 TAB 键，

因此使用 BC合入的版本大多会将缩进变乱。

9、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。

10、程序块的分界符（如C/C++语言的大括号‘{’和‘} ’）应各独占一行并且位于同一

列，同时与引用它们的语句左对齐。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。

示例：如下例子不符合规范。

for (...) {

... // program code

}

if (...)

{

... // program code

}

void example\_fun( void )

{

... // program code

}

应如下书写。

for (...)

{

... // program code

}

if (...)

{

... // program code

}

void example\_fun( void )

{

... // program code

}

11、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。

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。

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)

说明：一行程序以小于80字符(或者横向一屏)为宜，不要写得过长。

**二、 注释**

1、一般情况下，源程序**有效注释量必须在20％以上**。

说明：注释的原则是有助于对程序的阅读理解，在该加的地方都加了，注释不宜太多也不

能太少，注释语言必须准确、易懂、简洁。

2、说明性文件（如头文件.h文件、.inc文件、.def文件、编译说明文件.cfg等）头部应

进行注释，注释必须列出：版权说明、版本号、生成日期、作者、内容、功能、与其它文件的关系、修改日志等，头文件的注释中还应有函数功能简要说明。

示例：下面这段头文件的头注释比较标准，当然，并不局限于此格式，但上述信息建议要

包含在内。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Copyright (C), 1988-1999, Huawei Tech. Co., Ltd.

File name: // 文件名

Author: Version: Date: // 作者、版本及完成日期

Description: // 用于详细说明此程序文件完成的主要功能，与其他模块

// 或函数的接口，输出值、取值范围、含义及参数间的控

// 制、顺序、独立或依赖等关系

Others: // 其它内容的说明

Function List: // 主要函数列表，每条记录应包括函数名及功能简要说明

1. ....

History: // 修改历史记录列表，每条修改记录应包括修改日期、修改

// 者及修改内容简述

1. Date:

Author:

Modification:

2. ...

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

或者将其修改类似于如下：具体的项目大家可以自由增改，并最终决定。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 自动状态机类

\* 文 件 名：AtuoState.cpp

\* 主 要 类：CAtuoState

\* 说 明：串口通信数据处理类

\* 创 建 人：fjf

\* 创建时间：2009-09-18

\* 修 改 人：

\* 修改时间：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

3、源文件头部应进行注释，列出：版权说明、版本号、生成日期、作者、模块目的/功能、

主要函数及其功能、修改日志等。

示例：下面这段源文件的头注释比较标准，当然，并不局限于此格式，但上述信息建议要

包含在内。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Copyright (C), 1988-1999, Huawei Tech. Co., Ltd.

FileName: test.cpp

Author: Version : Date:

Description: // 模块描述

Version: // 版本信息

Function List: // 主要函数及其功能

1. -------

History: // 历史修改记录

<author> <time> <version > <desc>

David 96/10/12 1.0 build this moudle

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

说明：Description 一项描述本文件的内容、功能、内部各部分之间的关系及本文件与

其它文件关系等。History是修改历史记录列表，每条修改记录应包括修改日期、修改

者及修改内容简述。

4、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。

示例：下面这段函数的注释比较标准，当然，并不局限于此格式，但上述信息建议要包含

在内。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: // 函数名称

Description: // 函数功能、性能等的描述

Calls: // 被本函数调用的函数清单

Called By: // 调用本函数的函数清单

Table Accessed: // 被访问的表（此项仅对于牵扯到数据库操作的程序）

Table Updated: // 被修改的表（此项仅对于牵扯到数据库操作的程序）

Input: // 输入参数说明，包括每个参数的作

// 用、取值说明及参数间关系。

Output: // 对输出参数的说明。

Return: // 函数返回值的说明

Others: // 其它说明

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

5、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。

6、注释的内容要清楚、明了，含义准确，防止注释二义性。

说明：错误的注释不但无益反而有害。

7、避免在注释中使用缩写，特别是非常用缩写。

说明：在使用缩写时或之前，应对缩写进行必要的说明。

8、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。

示例：如下例子不符合规范。

例1：

/\* get replicate sub system index and net indicator \*/

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

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

例2：

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

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

/\* get replicate sub system index and net indicator \*/

应如下书写

/\* get replicate sub system index and net indicator \*/

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

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

9、对于所有有物理含义的变量、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，说明其物理含义。变量、常量、宏的注释应放在其上方相邻位置或右方。

示例：

/\* active statistic task number \*/

#define MAX\_ACT\_TASK\_NUMBER 1000

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

10、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。

示例：可按如下形式说明枚举/数据/联合结构。

/\* sccp interface with sccp user primitive message name \*/

enum SCCP\_USER\_PRIMITIVE

{

N\_UNITDATA\_IND, /\* sccp notify sccp user unit data come \*/

N\_NOTICE\_IND, /\* sccp notify user the No.7 network can not \*/

/\* transmission this message \*/

N\_UNITDATA\_REQ, /\* sccp user's unit data transmission request\*/

};

11、全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及

存取时注意事项等的说明。

示例：

/\* The ErrorCode when SCCP translate \*/

/\* Global Title failure, as follows \*/ // 变量作用、含义

/\* 0 － SUCCESS 1 － GT Table error \*/

/\* 2 － GT error Others － no use \*/ // 变量取值范围

/\* only function SCCPTranslate() in \*/

/\* this modual can modify it, and other \*/

/\* module can visit it through call \*/

/\* the function GetGTTransErrorCode() \*/ // 使用方法

BYTE g\_GTTranErrorCode;

12、注释与所描述内容进行同样的缩排。

说明：可使程序排版整齐，并方便注释的阅读与理解。

示例：如下例子，排版不整齐，阅读稍感不方便。

void example\_fun( void )

{

/\* code one comments \*/

CodeBlock One

软件编程规范总则 2 注释

仅供内部使用 12

/\* code two comments \*/

CodeBlock Two

}

应改为如下布局。

void example\_fun( void )

{

/\* code one comments \*/

CodeBlock One

/\* code two comments \*/

CodeBlock Two

}

13、将注释与其上面的代码用空行隔开。

示例：如下例子，显得代码过于紧凑。

/\* code one comments \*/

program code one

/\* code two comments \*/

program code two

应如下书写

/\* code one comments \*/

program code one

/\* code two comments \*/

program code two

14、对变量的定义和分支语句（条件分支、循环语句等）必须编写注释。

说明：这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释帮

助更好的理解程序，有时甚至优于看设计文档。

15、对于 switch语句下的case语句，如果因为特殊情况需要处理完一个case后进入下一

个case处理，必须在该case语句处理完、下一个case语句前加上明确的注释。

说明：这样比较清楚程序编写者的意图，有效防止无故遗漏 break语句。

示例（注意斜体加粗部分）：

case CMD\_UP:

ProcessUp();

break;

case CMD\_DOWN:

ProcessDown();

break;

case CMD\_FWD:

ProcessFwd();

if (...)

{

...

break;

}

else

{

ProcessCFW\_B(); // now jump into case CMD\_A

}

case CMD\_A:

ProcessA();

break;

case CMD\_B:

ProcessB();

break;

case CMD\_C:

ProcessC();

break;

case CMD\_D:

ProcessD();

break;

...

16、避免在一行代码或表达式的中间插入注释。

说明：除非必要，不应在代码或表达中间插入注释，否则容易使代码可理解性变差。

17、通过对函数或过程、变量、结构等正确的命名以及合理地组织代码的结构，使代码成为

自注释的。

说明：清晰准确的函数、变量等的命名，可增加代码可读性，并减少不必要的注释。

18、在代码的功能、意图层次上进行注释，提供有用、额外的信息。

说明：注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读

者理解代码，防止没必要的重复注释信息。

示例：如下注释意义不大。

/\* if receive\_flag is TRUE \*/

if (receive\_flag)

而如下的注释则给出了额外有用的信息。

/\* if mtp receive a message from links \*/

if (receive\_flag)

19、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。

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。

示例：参见如下例子。

if (...)

{

// program code

while (index < MAX\_INDEX)

{

// program code

} /\* end of while (index < MAX\_INDEX) \*/ // 指明该条 while语句结

束

} /\* end of if (...)\*/ // 指明是哪条if语句结束

软件编程规范总则 2 注释

仅供内部使用 15

20、注释格式尽量统一，建议使用“/\* …… \*/ ”。

21、注释应考虑程序易读及外观排版的因素，使用的语言若是中、英兼有的，建议多使用中

文，除非能用非常流利准确的英文表达。

说明：注释语言不统一，影响程序易读性和外观排版，出于对维护人员的考虑，建议使用

中文。

软件编程规范总则 3 标志符命名

仅供内部使用 16

三、标识符命名

**使用驼峰方法还是匈牙利或者LINUX的小写下划线方式或者其它方式，大家可以讨论。**

1、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。

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。

示例：如下单词的缩写能够被大家基本认可。

temp 可缩写为 tmp ;

flag 可缩写为 flg ;

statistic 可缩写为 stat ;

increment 可缩写为 inc ;

message 可缩写为 msg ;

2、命名中若使用特殊约定或缩写，则要有注释说明。

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。

3、自己特有的命名风格，要自始至终保持一致，不可来回变化。

说明：个人的命名风格，在符合所在项目组或产品组的命名规则的前提下，才可使用。（即

命名规则中没有规定到的地方才可有个人命名风格）。

4、对于变量命名，禁止取单个字符（如i 、j 、k...），建议除了要有具体含义外，还能

表明其变量类型、数据类型等，但i、j、k作局部循环变量是允许的。

说明：变量，尤其是局部变量，如果用单个字符表示，很容易敲错（如 i 写成 j ），而编

译时又检查不出来，有可能为了这个小小的错误而花费大量的查错时间。

示例：下面所示的局部变量名的定义方法可以借鉴。

int liv\_Width

其变量名解释如下：

l 局部变量（ Local ） （其它：g 全局变量（ Global）... ）

i 数据类型（ Interger ）

v 变量（ Variable ） （其它：c 常量（Const ）... ）

Width 变量含义

这样可以防止局部变量与全局变量重名。

5、命名规范必须与所使用的系统风格保持一致，并在同一项目中统一，比如采用UNIX的

全小写加下划线的风格或大小写混排的方式，不要使用大小写与下划线混排的方式，用作特殊标识如标识成员变量或全局变量的m\_和g\_，其后加上大小写混排的方式是允许的。

示例： Add\_User 不允许，add\_user 、AddUser、m\_AddUser 允许。

6、除非必要，不要用数字或较奇怪的字符来定义标识符。

示例：如下命名，使人产生疑惑。

#define \_EXAMPLE\_0\_TEST\_

#define \_EXAMPLE\_1\_TEST\_

void set\_sls00( BYTE sls );

应改为有意义的单词命名

#define \_EXAMPLE\_UNIT\_TEST\_

#define \_EXAMPLE\_ASSERT\_TEST\_

void set\_udt\_msg\_sls( BYTE sls );

7、在同一软件产品内，应规划好接口部分标识符（变量、结构、函数及常量）的命名，防

止编译、链接时产生冲突。

说明：对接口部分的标识符应该有更严格限制，防止冲突。如可规定接口部分的变量与常

量之前加上“模块”标识等。

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。

说明：下面是一些在软件中常用的反义词组。

add / remove begin / end create / destroy

insert / delete first / last get / release

increment / decrement put / get

add / delete lock / unlock open / close

min / max old / new start / stop

next / previous source / target show / hide

send / receive source / destination

cut / paste up / down

示例：

int min\_sum;

int max\_sum;

int add\_user( BYTE \*user\_name );

int delete\_user( BYTE \*user\_name );

9、除了编译开关 /头文件等特殊应用，应避免使用\_EXAMPLE\_TEST\_之类以下划线开始和

结尾的定义。

**四、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。

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。

示例：下列语句中的表达式

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

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

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

如果书写为

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。

2、避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的枚举或宏来代替。

示例：如下的程序可读性差。

if (Trunk[index].trunk\_state == 0)

{

Trunk[index].trunk\_state = 1;

... // program code

}

应改为如下形式。

#define TRUNK\_IDLE 0

#define TRUNK\_BUSY 1

if (Trunk[index].trunk\_state == TRUNK\_IDLE)

{

Trunk[index].trunk\_state = TRUNK\_BUSY;

... // program code

}

3、源程序中关系较为紧密的代码应尽可能相邻。代码如有可能，尽量按指定行（建议五行）左右增加一行空行。方便阅读，但如果六七行左右可以解决一个模块，则可以安排到一起。

说明：便于程序阅读和查找。

示例：以下代码布局不太合理。

rect.length = 10;

char\_poi = str;

rect.width = 5;

若按如下形式书写，可能更清晰一些。

rect.length = 10;

rect.width = 5; // 矩形的长与宽关系较密切，放在一起。

char\_poi = str;

4、不要使用难懂的技巧性很高的语句，除非很有必要时。

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。

示例：如下表达式，考虑不周就可能出问题，也较难理解。

\* stat\_poi ++ += 1;

\* ++ stat\_poi += 1;

应分别改为如下。

\*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; ”

5 变量、结构

5．1去掉没必要的公共变量。

说明：公共变量是增大模块间耦合的原因之一，故应减少没必要的公共变量以降低模块间

的耦合度。

5．2仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。

说明：在对变量声明的同时，应对其含义、作用及取值范围进行注释说明，同时若有必要

还应说明与其它变量的关系。

5．3明确公共变量与操作此公共变量的函数或过程的关系，如访问、修改及创建等。

说明：明确过程操作变量的关系后，将有利于程序的进一步优化、单元测试、系统联调以

及代码维护等。这种关系的说明可在注释或文档中描述。

示例：在源文件中，可按如下注释形式说明。

RELATION System\_Init Input\_Rec Print\_Rec Stat\_Score

Student Create Modify Access Access

Score Create Modify Access Access, Modify

注：RELATION 为操作关系；System\_Init、Input\_Rec、Print\_Rec、Stat\_Score

为四个不同的函数；Student、Score为两个全局变量；Create 表示创建，Modify 表

示修改， Access 表示访问。

其中，函数 Input\_Rec、Stat\_Score 都可修改变量 Score，故此变量将引起函数间较

大的耦合，并可能增加代码测试、维护的难度。

5．4当向公共变量传递数据时，要十分小心，防止赋与不合理的值或越界等现象发生。

说明：对公共变量赋值时，若有必要应进行合法性检查，以提高代码的可靠性、稳定性。

5．5防止局部变量与公共变量同名。

说明：若使用了较好的命名规则，那么此问题可自动消除。

5．6严禁使用未经初始化的变量作为右值。

说明：特别是在 C/C++ 中引用未经赋值的指针，经常会引起系统崩溃。

6、构造仅有一个模块或函数可以修改、创建，而其余有关模块或函数只访问的公共变量，

防止多个不同模块或函数都可以修改、创建同一公共变量的现象。

说明：降低公共变量耦合度。

7、使用严格形式定义的、可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密

切的变量。

说明：使用标准的数据类型，有利于程序的移植。

示例：如下例子（在 DOS 下BC3.1环境中），在移植时可能产生问题。

void main()

{

register int index; // 寄存器变量

\_AX = 0x4000; // \_AX 是BC3.1提供的寄存器“伪变量”

... // program code

}

8、结构的功能要单一，是针对一种事务的抽象。

说明：设计结构时应力争使结构代表一种现实事务的抽象，而不是同时代表多种。结构中

的各元素应代表同一事务的不同侧面，而不应把描述没有关系或关系很弱的不同事务的元

素放到同一结构中。

示例：如下结构不太清晰、合理。

typedef struct STUDENT\_STRU

{

unsigned char name[8]; /\* student's name \*/

unsigned char age; /\* student's age \*/

unsigned char sex; /\* student's sex, as follows \*/

/\* 0 - FEMALE; 1 - MALE \*/

unsigned char

teacher\_name[8]; /\* the student teacher's name \*/

unisgned char

teacher\_sex; /\* his teacher sex \*/

} STUDENT;

若改为如下，可能更合理些。

typedef struct TEACHER\_STRU

{

unsigned char name[8]; /\* teacher name \*/

unisgned char sex; /\* teacher sex, as follows \*/

/\* 0 - FEMALE; 1 - MALE \*/

} TEACHER;

typedef struct STUDENT\_STRU

{

unsigned char name[8]; /\* student's name \*/

unsigned char age; /\* student's age \*/

unsigned char sex; /\* student's sex, as follows \*/

/\* 0 - FEMALE; 1 - MALE \*/

unsigned int teacher\_ind; /\* his teacher index \*/

} STUDENT;

9、不要设计面面俱到、非常灵活的数据结构。

说明：面面俱到、灵活的数据结构反而容易引起误解和操作困难。

10、不同结构间的关系不要过于复杂。

说明：若两个结构间关系较复杂、密切，那么应合为一个结构。

示例：如下两个结构的构造不合理。

typedef struct PERSON\_ONE\_STRU

{

unsigned char name[8];

unsigned char addr[40];

unsigned char sex;

unsigned char city[15];

} PERSON\_ONE;

typedef struct PERSON\_TWO\_STRU

{

unsigned char name[8];

unsigned char age;

unsigned char tel;

} PERSON\_TWO;

由于两个结构都是描述同一事物的，那么不如合成一个结构。

typedef struct PERSON\_STRU

{

unsigned char name[8];

unsigned char age;

unsigned char sex;

unsigned char addr[40];

unsigned char city[15];

unsigned char tel;

} PERSON;

11、结构中元素的个数应适中。若结构中元素个数过多可考虑依据某种原则把元素组成不同

的子结构，以减少原结构中元素的个数。

说明：增加结构的可理解性、可操作性和可维护性。

示例：假如认为如上的 \_PERSON结构元素过多，那么可如下对之划分。

typedef struct PERSON\_BASE\_INFO\_STRU

{

unsigned char name[8];

unsigned char age;

unsigned char sex;

} PERSON\_BASE\_INFO;

typedef struct PERSON\_ADDRESS\_STRU

{

unsigned char addr[40];

unsigned char city[15];

unsigned char tel;

} PERSON\_ADDRESS;

typedef struct PERSON\_STRU

{

PERSON\_BASE\_INFO person\_base;

PERSON\_ADDRESS person\_addr;

} PERSON;

12、仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，并减少引起

误用现象。

说明：合理排列结构中元素顺序，可节省空间并增加可理解性。

示例：如下结构中的位域排列，将占较大空间，可读性也稍差。

typedef struct EXAMPLE\_STRU

{

unsigned int valid: 1;

PERSON person;

unsigned int set\_flg: 1;

} EXAMPLE;

若改成如下形式，不仅可节省 1 字节空间，可读性也变好了。

typedef struct EXAMPLE\_STRU

{

unsigned int valid: 1;

unsigned int set\_flg: 1;

PERSON person ;

} EXAMPLE;

13、结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用保留余地

（如预留一些空间等）。

说明：软件向前兼容的特性，是软件产品是否成功的重要标志之一。如果要想使产品具有

较好的前向兼容，那么在产品设计之初就应为以后版本升级保留一定余地，并且在产品升

级时必须考虑前一版本的各种特性。

½5-9：留心具体语言及编译器处理不同数据类型的原则及有关细节。

说明：如在 C 语言中，static 局部变量将在内存“数据区”中生成，而非 static 局部

变量将在“堆栈”中生成。这些细节对程序质量的保证非常重要。

14、编程时，要注意数据类型的强制转换。

说明：当进行数据类型强制转换时，其数据的意义、转换后的取值等都有可能发生变化，

而这些细节若考虑不周，就很有可能留下隐患。

15、对编译系统默认的数据类型转换，也要有充分的认识。

示例：如下赋值，多数编译器不产生告警，但值的含义还是稍有变化。

char chr;

unsigned short int exam;

chr = -1;

exam = chr; // 编译器不产生告警，此时exam 为0xFFFF 。

17、尽量减少没有必要的数据类型默认转换与强制转换。应该使用static\_cast<>之类的转换，目的是提供出错机制的检查。

18、合理地设计数据并使用自定义数据类型，避免数据间进行不必要的类型转换。

19、对自定义数据类型进行恰当命名，使它成为自描述性的，以提高代码可读性。注意其

命名方式在同一产品中的统一。

说明：使用自定义类型，可以弥补编程语言提供类型少、信息量不足的缺点，并能使程序

清晰、简洁。

示例：可参考如下方式声明自定义数据类型。

下面的声明可使数据类型的使用简洁、明了。

typedef unsigned char BYTE;

typedef unsigned short WORD;

typedef unsigned int DWORD;

下面的声明可使数据类型具有更丰富的含义。

typedef float DISTANCE;

typedef float SCORE;

20、当声明用于分布式环境或不同CPU 间通信环境的数据结构时，必须考虑机器的字节顺

序、使用的位域及字节对齐等问题 。

说明：比如 Intel CPU与68360 CPU，在处理位域及整数时，其在内存存放的“顺序”

正好相反。

示例：假如有如下短整数及结构。

unsigned short int exam;

typedef struct EXAM\_BIT\_STRU

{ /\* Intel 68360 \*/

unsigned int A1: 1; /\* bit 0 7 \*/

unsigned int A2: 1; /\* bit 1 6 \*/

unsigned int A3: 1; /\* bit 2 5 \*/

} EXAM\_BIT;

如下是 Intel CPU 生成短整数及位域的方式。

内存： 0 1 2 ... （从低到高，以字节为单位）

exam exam 低字节 exam 高字节

内存： 0 bit 1 bit 2 bit ... （字节的各“位”）

EXAM\_BIT A1 A2 A3

如下是 68360 CPU 生成短整数及位域的方式。

内存： 0 1 2 ... （从低到高，以字节为单位）

exam exam 高字节 exam 低字节

内存： 7 bit 6 bit 5 bit ... （字节的各“位”）

EXAM\_BIT A1 A2 A3

说明：在对齐方式下， CPU 的运行效率要快得多。

示例：如下图，当一个 long 型数（如图中 long1 ）在内存中的位置正好与内存的字边界

对齐时，CPU 存取这个数只需访问一次内存，而当一个 long 型数（如图中的 long2）在

内存中的位置跨越了字边界时，CPU 存取这个数就需要多次访问内存，如 i960cx 访问这

样的数需读内存三次（一个 BYTE、一个SHORT 、一个BYTE，由 CPU 的微代码执行，对

软件透明），所有对齐方式下 CPU 的运行效率明显快多了。

1 8 16 24 32

------- ------- ------- -------

| long1 | long1 | long1 | long1 |

------- ------- ------- -------

| | | | long2 |

------- ------- ------- --------

| long2 | long2 | long2 | |

------- ------- ------- --------

| ....

**六、 函数、过程**

1、对所调用函数的错误返回码要仔细、全面地处理。

2、明确函数功能，精确（而不是近似）地实现函数设计。

3、编写可重入函数时，应注意局部变量的使用（如编写C/C++ 语言的可重入函数时，应使

用auto即缺省态局部变量或寄存器变量）。

说明：编写 C/C++语言的可重入函数时，不应使用 static 局部变量，否则必须经过特殊

处理，才能使函数具有可重入性。

4、编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即P、V 操作）等手段对其加以保护。

说明：若对所使用的全局变量不加以保护，则此函数就不具有可重入性，即当多个进程调

用此函数时，很有可能使有关全局变量变为不可知状态。

示例：假设 Exam 是int 型全局变量，函数 Squre\_Exam 返回 Exam 平方值。那么如下

函数不具有可重入性。

unsigned int example( int para )

{

unsigned int temp;

Exam = para; // （\*\*）

temp = Square\_Exam( );

return temp;

}

此函数若被多个进程调用的话，其结果可能是未知的，因为当（\*\*）语句刚执行完后，另

外一个使用本函数的进程可能正好被激活，那么当新激活的进程执行到此函数时，将使

Exam 赋与另一个不同的 para 值，所以当控制重新回到“temp = Square\_Exam( )”

后，计算出的temp 很可能不是预想中的结果。此函数应如下改进。

unsigned int example( int para )

{

unsigned int temp;

[ 申请信号量操作] // 若申请不到“信号量”，说明另外的进程正处于

Exam = para; // 给Exam 赋值并计算其平方过程中（即正在使用此

temp = Square\_Exam( ); // 信号），本进程必须等待其释放信号后，才可继

[ 释放信号量操作] // 续执行。若申请到信号，则可继续执行，但其

// 它进程必须等待本进程释放信号量后，才能再使

// 用本信号。

return temp;

}

5、在同一项目组应明确规定对接口函数参数的合法性检查应由函数的调用者负责还是由接

口函数本身负责，缺省是由函数调用者负责。

说明：对于模块间接口函数的参数的合法性检查这一问题，往往有两个极端现象，即：要

么是调用者和被调用者对参数均不作合法性检查，结果就遗漏了合法性检查这一必要的处

理过程，造成问题隐患；要么就是调用者和被调用者均对参数进行合法性检查，这种情况

虽不会造成问题，但产生了冗余代码，降低了效率。

6、防止将函数的参数作为工作变量。

说明：将函数的参数作为工作变量，有可能错误地改变参数内容，所以很危险。对必须改

变的参数，最好先用局部变量代之，最后再将该局部变量的内容赋给该参数。

示例：下函数的实现不太好。

void sum\_data( unsigned int num, int \*data, int \*sum )

{

unsigned int count;

\*sum = 0;

for (count = 0; count < num; count++)

{

\*sum += data[count]; // sum 成了工作变量，不太好。

}

}

若改为如下，则更好些。

void sum\_data( unsigned int num, int \*data, int \*sum )

{

unsigned int count ;

int sum\_temp;

sum\_temp = 0;

for (count = 0; count < num; count ++)

{

sum\_temp += data[count];

}

\*sum = sum\_temp;

}

7、函数的规模尽量限制在200 行以内。 推荐120行左右，最好在60行以内。

说明：不包括注释和空格行。

8、一个函数仅完成一件功能。

9、为简单功能编写函数。

说明：虽然为仅用一两行就可完成的功能去编函数好象没有必要，但用函数可使功能明确

化，增加程序可读性，亦可方便维护、测试。

示例：如下语句的功能不很明显。

value = ( a > b ) ? a : b ;

改为如下就很清晰了。

int max (int a, int b)

{

return ((a > b) ? a : b);

}

value = max (a, b);

或改为如下。

#define MAX (a, b) (((a) > (b)) ? (a) : (b))

value = MAX (a, b);

10、不要设计多用途面面俱到的函数。

说明：多功能集于一身的函数，很可能使函数的理解、测试、维护等变得困难。

11、函数的功能应该是可以预测的，也就是只要输入数据相同就应产生同样的输出。

说明：带有内部“存储器”的函数的功能可能是不可预测的，因为它的输出可能取决于内

部存储器（如某标记）的状态。这样的函数既不易于理解又不利于测试和维护。在C/C++

语言中，函数的 static 局部变量是函数的内部存储器，有可能使函数的功能不可预测，

然而，当某函数的返回值为指针类型时，则必须是 STATIC 的局部变量的地址作为返回值，

若为 AUTO 类，则返回为错针。

示例：如下函数，其返回值（即功能）是不可预测的。

unsigned int integer\_sum( unsigned int base )

{

unsigned int index;

static unsigned int sum = 0; // 注意，是static 类型的。

// 若改为auto 类型，则函数即变为可预测。

for (index = 1; index <= base; index++)

{

sum += index;

}

return sum;

}

12、尽量不要编写依赖于其他函数内部实现的函数。

说明：此条为函数独立性的基本要求。由于目前大部分高级语言都是结构化的，所以通过

具体语言的语法要求与编译器功能，基本就可以防止这种情况发生。但在汇编语言中，由

于其灵活性，很可能使函数出现这种情况。

示例：如下是在 DOS 下TASM 的汇编程序例子。过程 Print\_Msg 的实现依赖于

Input\_Msg 的具体实现，这种程序是非结构化的，难以维护、修改。

... // 程序代码

proc Print\_Msg // 过程（函数）Print\_Msg

... // 程序代码

jmp LABEL

... // 程序代码

endp

proc Input\_Msg // 过程（函数）Input\_Msg

... // 程序代码

LABEL:

... // 程序代码

endp

13、避免设计多参数函数，不使用的参数从接口中去掉。

说明：目的减少函数间接口的复杂度。

14、非调度函数应减少或防止控制参数，尽量只使用数据参数。

说明：本建议目的是防止函数间的控制耦合。调度函数是指根据输入的消息类型或控制命

令，来启动相应的功能实体（即函数或过程），而本身并不完成具体功能。控制参数是指

改变函数功能行为的参数，即函数要根据此参数来决定具体怎样工作。非调度函数的控制

参数增加了函数间的控制耦合，很可能使函数间的耦合度增大，并使函数的功能不唯一。

示例：如下函数构造不太合理。

int add\_sub( int a, int b, unsigned char add\_sub\_flg )

{

if (add\_sub\_flg == INTEGER\_ADD)

{

return (a + b);

}

else

{

return (a ٛ b);

}

}

不如分为如下两个函数清晰。

int add( int a, int b )

{

return (a + b);

}

int sub( int a, int b )

{

return (a ٛ b);

}

15、检查函数所有参数输入的有效性。

16、检查函数所有非参数输入的有效性，如数据文件、公共变量等。

说明：函数的输入主要有两种：一种是参数输入；另一种是全局变量、数据文件的输入，

即非参数输入。函数在使用输入之前，应进行必要的检查。

17、函数名应准确描述函数的功能。

18、让函数在调用点显得易懂、容易理解。

19、避免函数中不必要语句，防止程序中的垃圾代码。

说明：程序中的垃圾代码不仅占用额外的空间，而且还常常影响程序的功能与性能，很可

能给程序的测试、维护等造成不必要的麻烦。

20、防止把没有关联的语句放到一个函数中。

说明：防止函数或过程内出现随机内聚。随机内聚是指将没有关联或关联很弱的语句放到

同一个函数或过程中。随机内聚给函数或过程的维护、测试及以后的升级等造成了不便，

同时也使函数或过程的功能不明确。使用随机内聚函数，常常容易出现在一种应用场合需

要改进此函数，而另一种应用场合又不允许这种改进，从而陷入困境。

在编程时，经常遇到在不同函数中使用相同的代码，许多开发人员都愿把这些代码提出来，

并构成一个新函数。若这些代码关联较大并且是完成一个功能的，那么这种构造是合理的，

否则这种构造将产生随机内聚的函数。

示例：如下函数就是一种随机内聚。

void Init\_Var( void )

{

Rect.length = 0;

Rect.width = 0; /\* 初始化矩形的长与宽 \*/

Point.x = 10;

Point.y = 10; /\* 初始化“点”的坐标 \*/

}

矩形的长、宽与点的坐标基本没有任何关系，故以上函数是随机内聚。

应如下分为两个函数：

void Init\_Rect( void )

{

Rect.length = 0;

Rect.width = 0; /\* 初始化矩形的长与宽 \*/

}

void Init\_Point( void )

{

Point.x = 10;

Point.y = 10; /\* 初始化“点”的坐标 \*/

}

21、如果多段代码重复做同一件事情，那么在函数的划分上可能存在问题。

说明：若此段代码各语句之间有实质性关联并且是完成同一件功能的，那么可考虑把此段

代码构造成一个新的函数。

22、减少函数本身或函数间的递归调用。

说明：递归调用特别是函数间的递归调用（如 A->B->C->A ），影响程序的可理解性；递

归调用一般都占用较多的系统资源（如栈空间）；递归调用对程序的测试有一定影响。故

除非为某些算法或功能的实现方便，应减少没必要的递归调用。

23、改进模块中函数的结构，降低函数间的耦合度，并提高函数的独立性以及代码可读性、

效率和可维护性。优化函数结构时，要遵守以下原则：

（1）不能影响模块功能的实现。

（2）仔细考查模块或函数出错处理及模块的性能要求并进行完善。

（3）通过分解或合并函数来改进软件结构。

（4）考查函数的规模，过大的要进行分解。

（5）降低函数间接口的复杂度。

（6）不同层次的函数调用要有较合理的扇入、扇出。

（7）函数功能应可预测。

（8）提高函数内聚。（单一功能的函数内聚最高）

说明：对初步划分后的函数结构应进行改进、优化，使之更为合理。

七、 程序效率

1、编程时要经常注意代码的效率。

说明：代码效率分为全局效率、局部效率、时间效率及空间效率。全局效率是站在整个系

统的角度上的系统效率；局部效率是站在模块或函数角度上的效率；时间效率是程序处理

输入任务所需的时间长短；空间效率是程序所需内存空间，如机器代码空间大小、数据空

间大小、栈空间大小等。

2、在保证软件系统的正确性、稳定性、可读性及可测性的前提下，提高代码效率。

**说明：不能一味地追求代码效率，而对软件的正确性、稳定性、可读性及可测性造成影响。**

3、局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响。

4、通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率。

说明：这种方式是解决软件空间效率的根本办法。

示例：如下记录学生学习成绩的结构不合理。

typedef unsigned char BYTE;

typedef unsigned short WORD;

typedef struct STUDENT\_SCORE\_STRU

BYTE name[8];

BYTE age;

BYTE sex;

BYTE class;

BYTE subject;

float score;

} STUDENT\_SCORE;

因为每位学生都有多科学习成绩，故如上结构将占用较大空间。应如下改进（分为两个结

构），总的存贮空间将变小，操作也变得更方便。

typedef struct STUDENT\_STRU

{

软件编程规范总则 8 程序效率

仅供内部使用 43

BYTE name[8];

BYTE age;

BYTE sex;

BYTE class;

} STUDENT;

typedef struct STUDENT\_SCORE\_STRU

{

WORD student\_index;

BYTE subject;

float score;

} STUDENT\_SCORE;

5、循环体内工作量最小化。

说明：应仔细考虑循环体内的语句是否可以放在循环体之外，使循环体内工作量最小，从

而提高程序的时间效率。

示例：如下代码效率不高。

for (ind = 0; ind < MAX\_ADD\_NUMBER; ind++)

{

sum += ind;

back\_sum = sum; /\* backup sum \*/

}

语句“back\_sum = sum;”完全可以放在 for 语句之后，如下。

for (ind = 0; ind < MAX\_ADD\_NUMBER; ind++)

{

sum += ind;

}

back\_sum = sum; /\* backup sum \*/

6、仔细分析有关算法，并进行优化。

7、仔细考查、分析系统及模块处理输入（如事务、消息等）的方式，并加以改进。

8、对模块中函数的划分及组织方式进行分析、优化，改进模块中函数的组织结构，提高程

序效率。

说明：软件系统的效率主要与算法、处理任务方式、系统功能及函数结构有很大关系，仅

在代码上下功夫一般不能解决根本问题。

9、编程时，要随时留心代码效率；优化代码时，要考虑周全。

10、不应花过多的时间拼命地提高调用不很频繁的函数代码效率。

说明：对代码优化可提高效率，但若考虑不周很有可能引起严重后果。

11、要仔细地构造或直接用汇编编写调用频繁或性能要求极高的函数。

说明：只有对编译系统产生机器码的方式以及硬件系统较为熟悉时，才可使用汇编嵌入方

式。嵌入汇编可提高时间及空间效率，但也存在一定风险。

12、在保证程序质量的前提下，通过压缩代码量、去掉不必要代码以及减少不必要的局部和

全局变量，来提高空间效率。

说明：这种方式对提高空间效率可起到一定作用，但往往不能解决根本问题。

13、在多重循环中，应将最忙的循环放在最内层。

说明：减少 CPU 切入循环层的次数。

示例：如下代码效率不高。

for (row = 0; row < 100; row++)

{

for (col = 0; col < 5; col++)

{

sum += a[row][col];

}

}

可以改为如下方式，以提高效率。

for (col = 0; col < 5; col++)

{

for (row = 0; row < 100; row++)

{

sum += a[row][col];

}

}

14、尽量减少循环嵌套层次。

15、避免循环体内含判断语句，应将循环语句置于判断语句的代码块之中。

说明：目的是减少判断次数。循环体中的判断语句是否可以移到循环体外，要视程序的具

体情况而言，一般情况，与循环变量无关的判断语句可以移到循环体外，而有关的则不可

以。

示例：如下代码效率稍低。

for (ind = 0; ind < MAX\_RECT\_NUMBER; ind++)

{

if (data\_type == RECT\_AREA)

{

area\_sum += rect\_area[ind];

}

else

{

rect\_length\_sum += rect[ind].length;

rect\_width\_sum += rect[ind].width;

}

}

因为判断语句与循环变量无关，故可如下改进，以减少判断次数。

if (data\_type == RECT\_AREA)

{

for (ind = 0; ind < MAX\_RECT\_NUMBER; ind++)

{

area\_sum += rect\_area[ind];

}

}

else

{

for (ind = 0; ind < MAX\_RECT\_NUMBER; ind++)

{

rect\_length\_sum += rect[ind].length;

rect\_width\_sum += rect[ind].width;

}

}

16、尽量用乘法或其它方法代替除法，特别是浮点运算中的除法。

说明：浮点运算除法要占用较多 CPU 资源。

示例：如下表达式运算可能要占较多CPU 资源。

#define PAI 3.1416

radius = circle\_length / (2 \* PAI);

应如下把浮点除法改为浮点乘法。

#define PAI\_RECIPROCAL (1 / 3.1416 ) // 编译器编译时，将生成具体浮点数

radius = circle\_length \* PAI\_RECIPROCAL / 2;

17、不要一味追求紧凑的代码。

说明：因为紧凑的代码并不代表高效的机器码。

八、 宏

**若无必须，不建议使用宏，替代的方法是使用常量或者内联函数。**

1、用宏定义表达式时，要使用完备的括号。

示例：如下定义的宏都存在一定的风险。

#define RECTANGLE\_AREA( a, b ) a \* b

#define RECTANGLE\_AREA( a, b ) (a \* b)

#define RECTANGLE\_AREA( a, b ) (a) \* (b)

正确的定义应为：

#define RECTANGLE\_AREA( a, b ) ((a) \* (b))

2、将宏所定义的多条表达式放在大括号中。

示例：下面的语句只有宏的第一条表达式被执行。为了说明问题，for 语句的书写稍

不符规范。

#define INTI\_RECT\_VALUE( a, b )\

a = 0;\

b = 0;

for (index = 0; index < RECT\_TOTAL\_NUM; index++)

INTI\_RECT\_VALUE( rect.a, rect.b );

正确的用法应为：

#define INTI\_RECT\_VALUE( a, b )\

{\

a = 0;\

b = 0;\

}

for (index = 0; index < RECT\_TOTAL\_NUM; index++)

{

INTI\_RECT\_VALUE( rect[index].a, rect[index].b );

}

3、使用宏时，不允许参数发生变化。

示例：如下用法可能导致错误。

#define SQUARE( a ) ((a) \* (a))

int a = 5;

int b;

b = SQUARE( a++ ); // 结果：a = 7 ，即执行了两次增1。

正确的用法是：

b = SQUARE( a );

a++; // 结果：a = 6 ，即只执行了一次增1。

**九、类**

1、类格式（Class Format）

**类的长度限制在两千行代码以内。极限不允许超过四千行。**

声明属性依次序是public:、protected:、private:，每次缩进1个空格（译者注，为什么不是两个呢？也有人提倡private 在前，对亍声明了哪些数据成员一目了然，迓有人提倡依逻辑关系将发量不操作放在一起，都有道理:-)）。

类声明的基本格式如下：

class MyClass : public OtherClass

{

public: // Note the 1 space indent!

MyClass(); // Regular 2 space indent.

explicit MyClass(int var);

~MyClass() {}

void SomeFunction();

void SomeFunctionThatDoesNothing()

{

}

void set\_some\_var(int var) { some\_var\_ = var; }

int some\_var() const { return some\_var\_; }

private:

bool SomeInternalFunction();

int some\_var\_;

int some\_other\_var\_;

DISALLOW\_COPY\_AND\_ASSIGN(MyClass);

};

注意：

1) 所以基类名应在80列限制下尽量不子类名放在同一行；

2) 关键词public:、protected:、private:要缩迕1个空格（注，MSVC多使用tab缩进，且这三个关键词没有缩进）；

3) 除第一个关键词（一般是public）外，其他关键词前空一行，如果类比较小的话也可以不空；

4) 返些关键词后不要空行；

5) public放在最前面，然后是protected和private；

2、 初始化列表（Initializer Lists）

构造函数初始化列表放在同一行或按四格缩进并排几行。

两种可以接受的初始化列表格式：

// When it all fits on one line:

MyClass::MyClass(int var) : some\_var\_(var), some\_other\_var\_(var + 1)

{

或

// When it requires multiple lines, indent 4 spaces, putting the colon on

// the first initializer line:

MyClass::MyClass(int var)

: some\_var\_(var), // 4 space indent

some\_other\_var\_(var + 1)

{ // lined up

...

DoSomething();

...

}

3、结构体和类（Structs vs. Classes）

**仅当叧有数据时使用struct，其它一概使用class**。

在C++中，关键字struct和class几乎含义等同，我们为其人为添加语义，以便为定义的数据类型合理选择使用哪个关键字。

struct被用在仅包含数据的消极对象（passive objects）上，可能包括有关联的常量，但没有存取数据成员以外的函数功能，而存叏功能通过直接访问实现而无需方法调用，这儿提到的方法是指只用于处理数据成员的，如构造函数、析构函数、Initialize()、Reset()、Validate()。

如果需要更多的函数功能，class更适吅，如果不确定的话，直接使用class。

如果不STL结合，对亍仿函数（functors）和特性（traits）可以丌用class而是使用struct。

注意：类和结极体的成员发量使用不同的命名规则。