引言

今天人们越来越明白软件设计更多地是一种工程，而不是一种个人艺术。由于大型产品的开发通常由很多的人协同作战，如果不统一编程规范，最终合到一起的程序，其可读性将较差，这不仅给代码的理解带来障碍，增加维护阶段的工作量，同时不规范的代码隐含错误的可能性也比较大。

BELL 实验室的研究资料表明，软件错误中 18%左右产生于概要设计阶段，15%左右产生于详细设计阶段，而编码阶段产生的错误占的比例则接近 50%；分析表明，编码阶段产生的错误当中，语法错误大概占 20%左右，而由于未严格检查软件逻辑导致的错误、函数（模块）之间接口错误及由于代码可理解度低导致优化维护阶段对代码的错误修改引起的错误则占了一半以上。

可见，提高软件质量必须降低编码阶段的错误率。如何有效降低编码阶段的错误呢？BELL 实验室的研究人员制定了详细的软件编程规范，并培训每一位程序员，最终的结果把编码阶段的错误降至 10%左右，同时也降低了程序的测试费用，效果相当显著。

本文从代码的可维护性（可读、可理解性、可修改性）、代码逻辑与效率、函数（模块）接口、可测试性四个方面阐述了软件编程规范，规范分成规则和建议两种，其中规则部分为强制执行项目，而建议部分则不作强制，可根据习惯取舍。

2. 编码规范

2.1. 排版风格

<规则 1> 程序块采用缩进风格编写，缩进为 4 个空格位。排版不混合使用空格和 TAB 键。

<规则 2> 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（如ー>），后不应加空格。

采用这种松散方式编写代码的目的是使代码更加清晰。例如：

(1) 逗号、分号只在后面加空格

```
printf("%d %d %d" , 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(lCurrentTime >= 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

```
*pApple = 'a';          // 内容操作"*"与内容之间
flag = !bIsEmpty;              // 非操作"!"与内容之间
p = &cMem;              // 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
```

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，如最内层的括号内侧(即左括号后面和右括号前面)不要加空格，因为在 C/C++语言中括号已经是最清晰的标志了。

另外，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。

最后，即使留空格，也不要连续留两个以上空格(为了保证缩进和排比留空除外)。

<规则 3> 函数体的开始，类的定义，结构的定义，if、for、do、while、switch 及 case 语句中的程序都应采用缩进方式，憑捸蛻}捰禀独占一行并且位于同一列，同时与引用它们的语句左对齐

例如下例不符合规范。

```
for ( ... ) {
    ... // 程序代码
}

if ( ... )
    {
    ... // 程序代码
    }

void DoExam( void )
    {
    ... // 程序代码
    }
```

应如下书写。

```
for ( ... )
{
    ... // 程序代码
}

if ( ... )
{
```

```
    ... // 程序代码
}

void DoExam( void )
{
    ... // 程序代码
}
```

<规则 4> 功能相对独立的程序块之间或 for、if、do、while、switch 等语句前后应加一空行。

例如以下例子不符合规范。

例一：

```
if ( ! ValidNi( ni ) )
{
    ... // 程序代码
}
nRepssnInd = SsnData[ index ].nRepssnIndex ;
nRepssnNi  = SsnData[ index ].ni ;
```

例二：

```
char *pContext;
int     nIndex;
long lCounter;
pContext = new (CString);
if(pContext == NULL)
{
        return FALSE;
}
```

应如下书写

例一：

```
if ( ! ValidNi( ni ) )
{
    ... // 程序代码
}

nRepssnInd = SsnData[ index ].nRepssnIndex ;
nRepssnNi  = SsnData[ index ].ni ;
```

例二：

```
char *pContext;
int    nIndex;
long lCounter;

pContext = new (CString);
if(pContext == NULL)
{
        return FALSE;
}
```

<规则 5> if、while、for、case、default、do 等语句自占一行。

示例：如下例子不符合规范。

```
if(pUserCR == NULL) return;
```

应如下书写：

```
if( pUserCR == NULL )
{
        return;
}
```

<规则 6> 若语句较长（多于 80 字符），可分成多行写，划分出的新行要进行适应的缩进，使排版整齐，语句可读。

```
    memset(pData->pData + pData->nCount, 0,
            (m_nMax - pData->nCount) * sizeof(LPVOID));

    CNoTrackObject* pValue =
            (CNoTrackObject*)_afxThreadData->GetThreadValue(m_nSlot);

for ( i = 0, j = 0 ; ( i < BufferKeyword[ WordIndex ].nWordLength )
                && ( j < NewKeyword.nWordLength ) ; i ++ , j ++ )
{
    ...  // 程序代码
}
```

<规则 7> 一行最多写一条语句。

示例：如下例子不符合规范。

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

都应书写成：

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

<规则 8> 对结构成员赋值，等号对齐。

示例：

```
rect.top        = 0;
rect.left    = 0;
rect.right      = 300;
rect.bottom     = 200;
```

<规则 9> #define 的各个字段对齐

以下示例不符合规范

```
#define  MAX_TASK_NUMBER 100
#define  LEFT_X    10
#define  BOTTOM_Y  400
```

应书写成：

```
#define  MAX_TASK_NUMBER        100
#define  LEFT_X                 10
#define  BOTTOM_Y             400
```

<规则 10> 不同类型的操作符混合使用时，使用括号给出优先级。

如本来是正确的代码：

```
if( year % 4 == 0 || year % 100 != 0 && year % 400 == 0 )
```

如果加上括号，则更清晰。

```
if((year % 4) == 0 || ((year % 100) != 0 && (year % 400) == 0))
```

2.2. 可理解性

2.2.1.注释

注释的原则是有助于对程序的阅读理解，注释不宜太多也不能太少，太少不利于代码理解，太多则会对阅读产生干扰，因此只在必要的地方才加注释，而且注释要准确、易懂、尽可能简洁。注释量一般控制在 30%到 50%之间。

<规则 1> 程序在必要的地方必须有注释，注释要准确、易懂、简洁。

例如如下注释意义不大。

```
/* 如果 bReceiveFlag 为 TRUE */
if ( bReceiveFlag == TRUE)
```

而如下的注释则给出了额外有用的信息。

```
/* 如果 mtp 从连接处获得一个消息*/
if ( bReceiveFlag == TURE)
```

<规则 2> 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。

例子 1

```
/* 获得系统指针和网络指针的副本 */


nRepssnInd = SsnData[ index ].nRepssnIndex ;
nRepssnNi  = SsnData[ index ].ni ;
```

例子 2

```
nRepssnInd = SsnData[ index ].nRepssnIndex ;
nRepssnNi  = SsnData[ index ].ni ;
/*获得系统指针和网络指针的副本 */
```

应如下书写

```
/*获得系统指针和网络指针的副本 */
nRepssnInd = SsnData[ index ].nRepssnIndex ;
nRepssnNi  = SsnData[ index ].ni ;
```

<规则 3> 对于所有的常量，变量，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，在声明时都必须加以注释，说明其含义。

示例：

```
/* 活动任务的数量 */
#define  MAX_ACT_TASK_NUMBER  1000

#define  MAX_ACT_TASK_NUMBER  1000   /*活动任务的数量 */

/* 带原始用户信息的 SCCP 接口 */
enum  SCCP_USER_PRIMITIVE
{
    N_UNITDATA_IND ,   /* 向 SCCP 用户报告单元数据已经到达   */
    N_UNITDATA_REQ ,   /* SCCP 用户的单元数据发送请求          */
} ;
```

<规则 4> 头文件、源文件的头部，应进行注释。注释必须列出：文件名、作者、目的、功能、修改日志等。

例如：

```
/*********************************************
文件名：
编写者：
编写日期：
简要描述：
修改记录：
*********************************************/
```

说明：撰蛞 枋鳎一项描述本文件的目的和功能等。摞薷募锹緒是修改日志列表，每条修改记录应包括修改日期、修改者及修改内容简述。

<规则 5> 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、修改日志等。

形式如下：

```
/************************************************
函数名称：
简要描述：          // 函数目的、功能等的描述
输入：                  // 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，
输出：                  // 输出参数的说明，  返回值的说明
修改日志：
************************************************/
```

对一些复杂的函数，在注释中最好提供典型用法。

<规则 6> 仔细定义并明确公共变量的含义、作用、取值范围及使用方法。

在对变量声明的同时，应对其含义、作用、取值范围及使用方法进行注释说明，同时若有必要还应说明与其它变量的关系。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。

示例：

```
/*  SCCP 转换时错误代码 */
/*  全局错误代码，含义如下  */    // 变量作用、含义
/*  0 — 成功    1 — GT 表错误    2 —GT 错误    其它值— 未使用      */    // 变量取值范围
```

<规则 7> 对指针进行充分的注释说明，对其作用、含义、使用范围、注意事项等说明清楚。

在对指针变量、特别是比较复杂的指针变量声明时，应对其含义、作用及使用范围进行注释说明，如有必要，还应说明其使用方法、注意事项等。

示例：

```
/* 学生记录列表的头指针 */
/*          当在此模块中创建该列表时，该头指针必须初始化，    */
/*           这样可以利用 GetListHead()获得这一列表。*/     //指针作用、含义
/*           该指针只在本模块使用，其它模块通过调用 GetListHead()获取*/
/*           当使用时必须保证它非空    */     //使用范围、方法
STUDENT_RECORD     *pStudentRecHead;
```

<规则 8> 对重要代码段的功能、意图进行注释，提供有用的、额外的信息。并在该代码段的结束处加一行注释表示该段代码结束。

示例：

```
        /* 可选通道的组合 */
        if ((gsmBCIe31->radioChReq >= DUAL_HR_RCR)
         && (gsmBCIe32->radioChReq >= DUAL_HR_RCR))
        {
            gsmBCIe31->radioChReq = FR_RCR;
                gsmBCIe32->radioChReq = FR_RCR;
        }
        else if ((gsmBCIe31->radioChReq >= DUAL_HR_RCR)
               && (gsmBCIe32->radioChReq == FR_RCR) )
        {
            gsmBCIe31->radioChReq = FR_RCR;
        }
        else if ((gsmBCIe31->radioChReq == FR_RCR)
```

```
  && (gsmBCIe32->radioChReq >= DUAL_HR_RCR))
        {
            gsmBCIe32->radioChReq = FR_RCR;
        }
        /* 本块结束 ( 可选通道组合 ) */
```

<规则 9> 在 switch 语句中，对没有 break 语句的 case 分支加上注释说明。

示例：

```
   switch(SubT30State)
   {
   case TA0:
       AT(CHANNEL, "AT+FCLASS=1\r", 0);
       if(T30Status != 0)
    {
            return(1);
    }
       InitFax();                /* 准备发送传真 */
       AT(CHANNEL, "ATD\r",-1);    /*发送 CNG ，接收 CED 和 HDLC 标志*/
       T1_Flg = 1;
       iResCode = 0;
        /* 没有 break; */

   case TA1:
       iResCode = GetModemMsg(CHANNEL);
        break;

   default:
        break;
   }
```

<规则 10> 维护代码时，要更新相应的注释，删除不再有用的注释。

保持代码、注释的一致性，避免产生误解。

2.2 命名

本文列出 Visual C++的标识符命名规范。

<规则 1> 标识符缩写

形成缩写的几种技术：

1) 去掉所有的不在词头的元音字母。如 screen 写成 scrn，primtive 写成 prmv。

2) 使用每个单词的头一个或几个字母。如 Channel Activation 写成 ChanActiv，Release Indication 写成 RelInd。

3) 使用变量名中每个有典型意义的单词。如 Count of Failure 写成 FailCnt。

4) 去掉无用的单词后缀 ing，ed 等。如 Paging Request 写成 PagReq。

5) 使用标准的或惯用的缩写形式（包括协议文件中出现的缩写形式）。如 BSIC(Base Station Identification Code)、MAP(Mobile Application Part)。

关于缩写的准则：

1) 缩写应该保持一致性。如 Channel 不要有时缩写成 Chan，有时缩写成 Ch。Length 有时缩写成 Len，有时缩写成 len。

2) 在源代码头部加入注解来说明协议相关的、非通用缩写。

3) 标识符的长度不超过 32 个字符。

<规则 2> 变量命名约定

参照匈牙利记法，即

[作用范围域前缀] + [前缀] + 基本类型 + 变量名

其中：

前缀是可选项，以小写字母表示；

基本类型是必选项，以小写字母表示；

变量名是必选项，可多个单词(或缩写)合在一起，每个单词首字母大写。

前缀列表如下：

前缀	意义	举例
g_	Global 全局变量	g_MyVar
m_	类成员变量 或 模块级变量	m_ListBox, m_Size
s_	static 静态变量	s_Count
h	Handle 句柄	hWnd
p	Pointer 指针	pTheWord
lp	Long Point 长指针	lpCmd
a	Array 数组	aErr

基本类型列表如下：

基本类型	意义	举例
b	Boolean 布尔	bIsOK
by	Byte 字节	byNum
c	Char 字符	cMyChar
i 或 n	Intger 整数	nTestNumber
u	Unsigned integer 无符号整数	uCount
ul	Unsigned Long 无符号长整数	ulTime
w	Word 字	wPara

dw　Double Word 双字　dwPara

l　Long 长型　lPara

f　Float 浮点数　fTotal

s　String 字符串　sTemp

sz　NULL 结束的字符串　szTrees

fn　Funtion 函数　fnAdd

enm　枚举型　enmDays

x,y　x,y 坐标

<规则 3> 宏和常量的命名

宏和常量的命名规则：单词的字母全部大写，各单词之间用下划线隔开。命名举例：

```
#define  MAX_SLOT_NUM        8
#define  EI_ENCR_INFO        0x07

const int MAX_ARRAY
```

<规则 4> 结构和结构成员的命名

结构名各单词的字母均为大写，单词间用下划线连接。可用或不用 typedef，但是要保持一致，不能有的结构用 typedef，有的又不用。如：

```
typedef struct LOCAL_SPC_TABLE_STRU
{
     char    cValid;
     int     nSpcCode[MAX_NET_NUM];
} LOCAL_SPC_TABLE ;
```

结构成员的命名同变量的命名规则。

<规则 5> 枚举和枚举成员的命名

枚举名各单词的字母均为大写，单词间用下划线隔开。

枚举成员的命名规则：单词的字母全部大写，各单词之间用下划线隔开；要求各成员的第一个单词相同。命名举例：

```
typdef enum
{
LAPD_ MDL_ASSIGN_REQ,
LAPD_MDL_ASSIGN_IND,
LAPD_DL_DATA_REQ,
LAPD_DL_DATA_IND,
LAPD_DL_UNIT_DATA_REQ,
LAPD_DL_UNIT_DATA_IND,
```

```
} LAPD_PRMV_TYPE;
```

<规则 6> 类的命名

前缀	意义	举例
C	类	CMyClass
CO	COM 类	COMMyObjectClass
CF	COM class factory	CFMyClassFactory
I	COM interface class	IMyInterface
CImpl	COM implementation class	CImplMyInterface

<规则 7> 函数的命名

单词首字母为大写，其余均为小写，单词之间不用下划线。函数名应以一个动词开头，即函数名应类似搅 鼋峁箔。命名举例：

```
void PerformSelfTest(void) ;
void ProcChanAct(MSG_CHAN_ACTIV *pMsg, UC MsgLen);
```

2.3. 可维护性

<规则 1> 在逻辑表达式中使用明确的逻辑判断。

示例：如下逻辑表达式不规范。

```
1)  if ( strlen(strName) )
2)  for ( index = MAX_SSN_NUMBER; index ; index -- )
3)  while ( p && *p )   // 假设 p 为字符指针
```

应改为如下：

```
1)  if ( strlen(strName) != 0 )
2)  for ( index = MAX_SSN_NUMBER; index != 0 ; index -- )
3)  while ((p != NULL) && (*p != '\0' ))
```

<规则 2> 预编译条件不应分离一完整的语句。

不正确：

```
if (( cond == GLRUN)
#ifdef DEBUG
      || (cond == GLWAIT)
#endif
)
{

}
```

正确：

```
#ifdef DEBUG
if( cond == GLRUN || cond == GLWAIT )
#else
if( cond == GLRUN )
#endif
{

}
```

<规则 3> 在宏定义中合并预编译条件。

不正确：

```
#ifdef  EXPORT
    for ( i = 0; i < MAX_MSXRSM; i++ )
#else
for ( i = 0; i < MAX_MSRSM; i++ )
#endif
```

正确：

头文件中：

```
#ifdef  EXPORT
#define MAX_MS_RSM    MAX_MSXRSM
#else
#define MAX_MS_RSM    MAX_MSRSM
#endif
```

源文件中：

```
for( i = 0; i < MAX_MS_RSM; i++ )
```

<规则 4> 使用宏定义表达式时，要使用完备的括号。

如下的宏定义表达式都存在一定的隐患。

```
#define  REC_AREA(a, b)        a * b
#define  REC_AREA(a, b)         (a * b)
#define  REC_AREA(a, b)         (a) * (b)
```

正确的定义为：

```
#define  REC_AREA(a, b)         ((a) * (b))
```

<规则 5> 宏所定义的多条表达式应放在大括号内。

示例：下面的语句只有宏中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  INIT_RECT_VALUE( a, b )          a = 0 ;          b = 0 ;

for ( index = 0 ; index < RECT_TOTAL_NUM ; index ++ )
    INIT_RECT_VALUE( rect.a,  rect.b ) ;
```

正确的用法应为：

```
#define  INIT_RECT_VALUE( a, b )          {                        a =
0 ;              b = 0 ;          }

for ( index = 0 ; index < RECT_TOTAL_NUM ; index ++ )
{
    INIT_RECT_VALUE( rect[ index ].a,  rect[ index ].b ) ;
}
```

<规则 6> 宏定义不能隐藏重要的细节，避免有 return，break 等导致程序流程转向的语句。

如下例子是不规范的应用，其中隐藏了程序的执行流程。

```
#define FOR_ALL        for(i = 0; i < SIZE; i++)

/* 数组 c 置 0 */
FOR_ALL
{
    c[i] = 0;
}

    #define CLOSE_FILE    {                                fclose
(fp_local);                                fclose
(fp_urban);
return;                                        }
```

<规则 7> 使用宏时，不允许参数发生变化。

下面的例子隐藏了重要的细节，隐含了错误。

```
#define SQUARE    ((x) * (x))
                .
                .
```

.

```
w = SQUARE(++value);
```

这个引用将被展开称：

```
w = ((++value) * (++value));
```

其中 value 累加了两次，与设计思想不符。正确的用法是：

```
w = SQUARE(x);
x++;
```

<规则 8> 当 if、while、for 等语句的程序块为摽諒时，使用擲}敳　拧_

```
while ( *s++ == *t++ ) ;
```

以上代码不符合规范，正确的书写方式为：

```
while( *s++ == *t++ )
{
    /* 无循环体 */
}
```

或

```
while( *s++ == *t++ )
{
}
```

<规则 9> 结构中元素布局合理，一行只定义一个元素。

如下例子不符合规范，

```
typedef struct
{
    _UI    left, top, right, bottom;
} RECT;
```

应书写称：

```
typedef struct
{
    _UI    left;        /* 矩形左侧 x 坐标 */
    _UI top;
    _UI right;
    _UI bottom;
} RECT;
```

<规则 10> 枚举值从小到大顺序定义。

<规则 11> 包含头文件时，使用撓喽月肪稊，不使用摆　月肪稊。

如下引用：

```
#include     "c:\switch\inc\def.inc"
```

应改为：

```
#include "inc\def.inc"
```

或

```
#include "def.inc"
```

<规则 12> 不允许使用复杂的操作符组合等。

下面用法不好，

```
iMaxVal = ( (a > b ? a : b) > c ? (a > b ? a : b) : c );
```

应该为：

```
iTemp    = ( a > b ? a : b);
iMaxVal = (iTemp > b ? iTemp : b);
```

不要把"++"、"--"操作符与其他如"+="、"-="等组合在一起形成复杂奇怪的表达式。如下的表达式那以理解。

```
*pStatPoi++ += 1;
*++pStatPoi += 1;
```

应分别改为：

```
*pStatPoi += 1;
pStatPoi++;
            和
            ++pStatPoi;
*pStatPoi += 1;
```

<规则 13> 函数和过程中关系较为紧密的代码尽可能相邻。

如初始化代码应放在一起，不应在中间插入实现其它功能的代码。以下代码不符合规范，

```
for (uiUserNo = 0; uiUserNo < MAX_USER_NO; uiUserNo++)
{
    ...;    /* 初始化用户数据 */
}

pSamplePointer = NULL;
g_uiCurrentUser = 0;        /* 设置当前用户索引号 */
```

应必为：

```
for (uiUserNo = 0; uiUserNo < MAX_USER_NO; uiUserNo++)
{
    ...;    /* 初始化用户数据 */
}

g_uiCurrentUser = 0;        /* 设置当前用户索引号 */

pSamplePointer = NULL;
```

<规则 14> 每个函数的源程序行数原则上应该少于 200 行。

对于消息分流处理函数，完成的功能统一，但由于消息的种类多，可能超过 200 行的限制，不属于违反规定。

<规则 15> 语句嵌套层次不得超过 5 层。

嵌套层次太多，增加了代码的复杂度及测试的难度，容易出错，增加代码维护的难度。

<规则 16> 用 sizeof 来确定结构、联合或变量占用的空间。

这样可提高程序的可读性、可维护性，同时也增加了程序的可移植性。

<规则 17> 避免相同的代码段在多个地方出现。

当某段代码需在不同的地方重复使用时，应根据代码段的规模大小使用函数调用或宏调用的方式代替。这样，对该代码段的修改就可在一处完成，增强代码的可维护性。

<规则 18> 使用强制类型转换。

示例：

```
USER_RECORD    *pUser;

pUser = (USER_RECORD *) malloc (MAX_USER * sizeof(USER_RECORD));
```

<规则 19> 避免使用 goto 语句。

<规则 20> 避免产生拏绦蚪釘（program knots），在循环语句中，尽量避免 break、goto 的使用。

如下例子：

```
for( i = 0; i < n; i++)
{
        bEof = fscanf( pInputFile, "%d;", &x[i]);
        if( bEof == EOF )
```

```
        {
            break;
        }
        nSum += x[i];
}
```

最好按以下方式书写，避免程序打掺釘：

```
for( i = 0; i < n && bEof= EOF; i++)
{
        bEof = fscanf( pInputFile, "%d;", &x[i]);
        if( bEof!= EOF )
        {
        nSum += x[i];
        }
}
```

<规则 21> 功能相近的一组常量最好使用枚举来定义。

不推荐定义方式：

```
/* 功能寄存器值 */
#define  ERR_DATE        1        /* 日期错误 */
#define  ERR_TIME        2        /* 时间错误 */
#define  ERR_TASK_NO    3        /* 任务号错误 */
```

推荐按如下方式书写：

```
/*功能寄存器值 */
enum ERR_TYPE
{
        ERR_DATE        =    1,            /*日期错误 */
        ERR_TIME        =    2,            /*时间错误 */
        ERR_TASK_NO    =    3                    /* 任务号错误 */
}
```

<规则 22> 每个函数完成单一的功能，不设计多用途面面俱到的函数。

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。

使函数功能明确化，增加程序可读性，亦可方便维护、测试。

<建议 1> 循环、判断语句的程序块部分用花括号括起来，即使只有一条语句。

如：

```
if( bCondition == TRUE )
      bFlag = YES;
```

建议按以下方式书写：

```
if( bCondition == TRUE )
{
        bFlag = YES;
}
```

这样做的好处是便于代码的修改、增删。

<建议 2> 一行只声明一个变量。

不推荐的书写方式：

```
void DoSomething(void)
{
    int Amicrtmrs,  nRC;
    int nCode, nStatus;
```

推荐做法：

```
void DoSomething(void)
{
    int nAmicrtmrs;        /* ICR 计时器 */
    int nRC;                /* 返回码 */
    int nCode;             /* 访问码 */
    int nStatus;            /* 处理机状态 */
```

<建议 3> 使用专门的初始化函数对所有的公共变量进行初始化。

<建议 4> 使用可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密切的变量。

<建议 5> 用明确的函数实现不明确的语句功能

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 ) ;
```

4.　程序正确性、效率

<规则 1> 严禁使用未经初始化的变量。

引用未经初始化的变量可能会产生不可预知的后果，特别是引用未经初始化的指针经常会导致系统崩溃，需特别注意。声明变量的同时初始化，除了能防止引用未经初始化的变量外，还可能生成更高效的机器代码。

<规则 2> 定义公共指针的同时对其初始化。

这样便于指针的合法性检查，防止应用未经初始化的指针。建议对局部指针也在定义的同时初始化，形成习惯。

<规则 3> 较大的局部变量（2K 以上）应声明成静态类型（static），避免占用太多的堆栈空间。

避免发生堆栈溢出，出现不可预知的软件故障。

<规则 4> 防止内存操作越界。

说明：内存操作主要是指对数组、指针、内存地址等的操作。内存操作越界是软件系统主要错误之一，后果往往非常严重，所以当我们进行这些操作时一定要仔细小心。

A. 数组越界。

```
char aMyArray[10];
for( i = 0; i <= 10; i++ )
{
    aMyArray[i] = 0;     //当 i 等于 10 时，将发生越界。
}
```

B. 指针操作越界。

```
char aMyArray[10];
char *pMyArray;
```

```
pMyArray = aMyArray;
--pMyArray;          // 越界

pMyArray = aMyArray;
pMyArray += 10;     // 越界
```

<规则 5> 减少没必要的指针使用，特别是较复杂的指针，如指针的指针、数组的指针，指针的数组，函数的指针等。

用指针虽然灵活，但也对程序的稳定性造成一定威胁，主要原因是当要操作一个指针时，此指针可能正指向一个非法的地址。安安全全地使用一个指针并不是一件容易的事情。

<规则 6> 防止引用已经释放的内存空间。

在实际编程过程中，稍不留心就会出现在一个模块中释放了某个内存块（如指针），而另一模块在随后的某个时刻又使用了它。要防止这种情况发生。

<规则 7> 程序中分配的内存、申请的文件句柄，在不用时应及时释放或关闭。

分配的内存不释放以及文件句柄不关闭，是较常见的错误，而且稍不注意就有可能发生。这类错误往往会引起很严重后果，且难以定位。

<规则 8> 注意变量的有效取值范围，防止表达式出现上溢或下溢。

示例：

```
unsigned char cIndex = 10;
while( cIndex--  >= 0 )
{
}         //将出现下溢
```

当 cIndex 等于 0 时，再减 1 不会小于 0，而是 0xFF，故程序是一个死循环。

```
char chr = 127;
chr += 1;          //127 为 chr 的边界值，再加 1 将使 chr 上溢到-128，而不是 128。
```

<规则 9> 防止精度损失。

以下代码将产生精度丢失。

```
#define DELAY_MILLISECONDS     10000
char time;
time = DELAY_MILLISECONDS;
WaitTime( time );
```

代码的本意是想产生 10 秒钟的延时，然而由于 time 为字符型变量，只取 DELAY_MILLISECONDS 的低字节，高位字节将丢失，结果只产生了 16 毫秒的延时。

<规则 10> 防止操易混淆的作符拼写错误。

形式相近的操作符最容易引起误用，如 C/C++中的“=斟鬲==敗　|斟鬲||敗　&斟鬲&&數齲

羝葱创砹耍　嘁肫鞑灰欢　茨患觳槌隼础_

示例：如把“&孱闯堇&&敫　蚍粗　_

```
bRetFlag = ( pMsg -> bRetFlag & RETURN_MASK ) ;
```

被写为：

```
bRetFlag = ( pMsg -> bRetFlag && RETURN_MASK ) ;
```

<规则 11> 使用无符号类型定义位域变量。

示例：

```
typedef struct
{
        int bit1 : 1;
        int bit2 : 1;
        int bit3 : 1;
} bit;

            bit.bit1 = 1;
    bit.bit2 = 3;
            bit.bit3 = 6;
            printf("%d, %d, %d", bit.bit1, bit.bit2, bit.bit3 );
```

输出结果为：-1,-1, -2，不是：1,3,6.

<规则 12> switch 语句的程序块中必须有 default 语句。

对不期望的情况（包括异常情况）进行处理，保证程序逻辑严谨。

<规则 13> 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，使用的位域也要有充分的考虑。

比如 Intel CPU 与 68360 CPU，在处理位域及整数时，其在内存存放的撂承驍，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      2   */
    unsigned int A2 : 1 ;  /* bit  1      1   */
    unsigned int A3 : 1 ;  /* bit  2      0   */
} _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　...　（字节的各撐粑）

_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：　　0 bit　　1 bit　　2 bit　...　（字节的各撐粑）

_EXAM_BIT　　A3　　A2　　A1

<规则 14> 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）。

可重入性是指函数可以被多个任务进程调用。在多任务操作系统中，函数是否具有可重入性是非常重要的，因为这是多个进程可以共用此函数的必要条件。另外，编译器是否提供可重入函数库，与它所服务的操作系统有关，只有操作系统是多任务时，编译器才有可能提供可重入函数库。如 DOS 下 BC 和 MSC 等就不具备可重入函数库，因为 DOS 是单用户单任务操作系统。

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。

<规则 15> 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。

<规则 16> 结构中的位域应尽可能相邻。结构中的位域在开始处应对齐撟纸趿或撟謹的边界。

这样可减少结构占用的内存空间，减少 CPU 处理位域的时间，提高程序效率。

示例：如下结构中的位域布局不合理。（假设例子在 Intel CPU 环境下）

```
typedef struct _EXAMPLE_STRU
{
    unsigned int nExamOne   : 6 ;
    unsigned int nExamTwo   : 3 ;  // 此位域跨越字节掺唤訑处。
    unsigned int nExamThree : 4 ;
} _EXAMPLE ;
```

应改为如下（按字节对齐）。

```
typedef struct _EXAMPLE_STRU
{
    unsigned int nExamOne   : 6 ;
    unsigned int nFreeOne   : 2 ;  // 保留 bit 位，使下个位域从字节开始。
    unsigned int nExamTwo   : 3 ;  // 此位域从新的字节处开始。
    unsigned int nExamThree : 4 ;
} _EXAMPLE ;
```

<规则 17> 避免函数中不必要语句，防止程序中的垃圾代码，预留代码应以注释的方式出现。

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。

<规则 18> 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。

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  _UC ;
typedef unsigned int   _UI ;

typedef struct _STUDENT_SCORE_STRU
{
    _UC  szName[ 8 ] ;
    _UC  cAge ;
    _UC  cSex ;
    _UC  cClass ;
    _UC  cSubject ;
    float fScore ;
} _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
{
    _UC  szName[ 8 ] ;
    _UC  cAge ;
```

```
    _UC  cSex ;
    _UC  cClass ;
} _STUDENT ;

typedef struct _STUDENT_SCORE_STRU
{
    _UI  iStudentIndex ;
    _UC  cSubject ;
    float  fScore ;
} _STUDENT_SCORE ;
```

<规则 19> 循环体内工作量最小化。

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 ( i= 0 ; i< MAX_ADD_NUMBER ; i++ )
{
    nSum += i;
    nBackSum  = nSum ;  /* 备份和 */
}
```

语句搀 BackSum = nSum ;亹者 梢苑旁趣 or 语句之后，如下。

```
for ( i = 0 ; i < MAX_ADD_NUMBER ; i ++ )
{
    nSum += i ;
}
nBackSum  = nSum ;  /*备份和 */
```

<规则 20> 在多重循环中，应将最忙的循环放在最内层。

<规则 21> 避免循环体内含判断语句，将与循环变量无关的判断语句移到循环体外。

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。

<规则 22> 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，在时间效率要求不是特别严格时，要优先保证程序的可读性。

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
fRadius = fCircleLength / ( 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
fRadius = fCircleLength * PAI_RECIPROCAL / 2 ;
```

<规则 23> 用“++敞莹--敨僮鞔　鎒+=1 敞莹-=1 敞　峋叱绦蛩俣取_

<规则 24> 系统输入（如用户输入）、系统输出（如信息包输出）、系统资源操作（如内存分配、文件及目录操作）、网络操作（如通信、调用等）、任务之间的操作（如通信、调用等）时必须进行错误、超时或者异常处理。

<建议 1> 定义字符串变量的同时将其初始化为空即摄，以避免无限长字符串。

<建议 2> 在 switch 语句中将经常性的处理放在前面。

2.5.　接口

<规则 1> 头文件应采用 #ifndef / #define / #endif 的方式来防止多次被嵌入。

示例如下：

假设头文件为摘 EF.INC"，则其内容应为：

```
#ifndef __DEF_INC
#define __DEF_INC
...
#endif
```

<规则 2> 去掉没有必要的公共变量，编程时应尽量少用公共变量。

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。应该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。

<规则 3> 当向公共变量传递数据时，要防止越界现象发生。

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。

<规则 4> 返回值为指针的函数，不可将局部变量的地址作为返回值。

当函数退出时，非 static 局部变量将消失，所以引用返回的指针将可能引起严重后果。下例将不能完成正确的功能。

```
char *GetFilename(int nFileNo)
{
```

```
        char szFileName[20];

        sprintf( szFileName, "COUNT%d", nFileNo);
        return szFileName;
}
```

<规则 5> 尽量不设计多参数函数，将不使用的参数从接口中去掉，降低接口复杂度。
减少函数间接口的复杂度。

<规则 6> 对所调用函数的返回码要仔细、全面地处理。
防止把错误传递到后面的处理流程。如有意不检查其返回码，应明确指明。
如：

```
(void)fclose(fp);
```

<规则 7> 显示地给出函数的返回值类型。无返回值函数定义为 void。
C、C++语言的编译系统默认无显示返回值函数的返回值类型为 int。

<规则 8> 声明函数原型时给出参数名称和类型，并且与实现此函数时的参数名称、类型保持一致，无参数的函数，用 void 声明。
示例：下面声明不正确。

```
int  CheckData( ) ;
int  SetPoint( int, int ) ;
int  SetPoint( x, y )
int x, y;
```

应改为如下声明：

```
int  CheckData( void ) ;
int  SetPoint( int x, int y ) ;
```

<规则 9> 检查接口函数所有输入参数的有效性。
可直接检查或使用断言进行检查，尤其是指针参数。只在本模块内使用的函数可不检查。

<规则 10> 检查函数的所有非参数输入，如数据文件、公共变量等。
可直接检查或使用断言进行检查，尤其是指针变量。

<规则 11> 声明函数原型时，对于数组型参数，不要声明为指针，维护函数接口的清晰性。
示例：假设函数 SortInt()完成的功能是对一组整数排序，接受的参数是一整数数

组及数组中的元素个数，以下声明不符合规范。

```
void SortInt(int num, int *data);
```

应声明为：

```
void SortInt(int num, int data[]);
```

2.6.代码可测性

<规则 1> 模块编写应该有完善的测试方面的考虑。

<规则 2> 源代码中应该设计了代码测试的内容，如打印宏开关、变量值、函数名称、函数值等。

在编写代码之前，应预先设计好程序调试与测试的方法和手段，并设计好各种调测开关及相应测试代码如打印函数等。

程序的调试与测试是软件生存周期中很重要的一个阶段，如何对软件进行较全面、高率的测试并尽可能地找出软件中的错误就成为很关键的问题。因此在编写源代码之前，除了要有一套比较完善的测试计划外，还应设计出一系列代码测试手段，为单元测试、集成测试及系统联调提供方便。

<规则 3> 在同一项目组或产品组内，要有一套统一的为集成测试与系统联调准备的调测开关及相应打印函数，并且要有详细的说明。

本规则是针对项目组或产品组的。

示例：.ext 文件示例，文件名为：EXAMPLE.EXT。

```
/* 头文件开始 */
#ifndef  __EXAMPLE_EXT
#define  __EXAMPLE_EXT

#define  _EXAMPLE_DEBUG_     // 模块测试总开关。打开开关的含义是模块可以
                             // 进行单元测试或其它功能、目的等的测试。
#ifdef  _EXAMPLE_DEBUG_
    #define  _EXAMPLE_UNIT_TEST_     // 单元测试宏开关
    #define  _EXAMPLE_ASSERT_TEST_   // 断言测试开关
    ...                              // 其它测试开关
#endif

#ifndef  _EXAMPLE_UNIT_TEST_  // 若没有定义单元测试
```

```
    #include <common.h>    // 各模块共用的头文件
    #include <os.h>        // 系统接口头文件

    #ifndef  _SYSTEM_DEBUG_VERSION_  // 如果是发行版本（即非 DEBUG 版）
        #undef  _EXAMPLE_UNIT_TEST_
        #undef  _EXAMPLE_ASSERT_TEST_
        ...     // 将所有与测试有关的开关都关掉，即编译时不含任何测试代码
    #endif

    #include <module.h>    // 与另一模块的接口头文件
    ...                    // 其它接口头文件
#else  // 若定义了单元测试，则应构造单元测试所需的环境、结构等。
    typdef unsigned char  _UC ;
    typdef unsigned long  _UL ;
    #define TRUE   1
    ...      // 所有为单元测试准备的环境，如宏、枚举、结构、联合等。
#endif

#endif  /*  EXAMPLE.EXT 结束 */
/* 头文件结束 */
```

<规则 4> 在同一项目组或产品组内，调测打印出的信息串的格式要有统一的形式。信息串中至少要有所在模块名（或源文件名）及行号。

统一的调测信息格式便于集成测试。

<规则 5> 使用断言来发现软件问题，提高代码可测性。

断言是对某种假设条件进行检查（可理解为若条件成立则无动作，否则应报告），它可以快速发现并定位软件问题，同时对系统错误进行自动报警。断言可以对在系统中隐藏很深，用其它手段极难发现的问题进行定位，从而缩短软件问题定位时间，提高系统的可测性。实际应用时，可根据具体情况灵活地设计断言。

示例：下面是 C 语言中的一个断言，用宏来设计的。（其中 NULL 为 0L）

```
#ifdef _EXAM_ASSERT_TEST_      // 若使用断言测试

void ExamAssert( char * szFileName, unsigned int nLineNo )
{
    printf( "\n[EXAM] Assert failed: %s, line %u\n",
            szFileName, nLineNo ) ;
    abort( ) ;
```

```
}

#define  EXAM_ASSERT( condition )          if ( condition )         \        //
若条件成立，则无动作
             NULL ;                        else                     \        //
否则报告
             ExamAssert( __FILE__, __LINE__ )

#else    // 若不使用断言测试

#define EXAM_ASSERT( condition )  NULL

#endif  /* ASSERT 结束 */
```

<规则 6> 用断言来检查程序正常运行时不应发生但在调测时有可能发生的非法情况。

<规则 7> 不能用断言代替错误处理来检查最终产品肯定会出现且必须处理的错误情况。
如某模块收到其它模块或链路上的消息后，要对消息的合理性进行检查，此过程为正常的错误检查，不能用断言来代替。

<规则 8> 用断言确认函数的参数。
示例：假设某函数参数中有一个指针，那么使用指针前可对它检查，如下。

```
int ExamFunc( unsigned char *str )
{
    EXAM_ASSERT( str != NULL ) ;  // 用断言检查撵偕柚刚氩晃  諒这个条件

    ... // 其它程序代码
}
```

<规则 9> 用断言保证没有定义的特性或功能不被使用。
示例：假设某通信模块在设计时，准备提供撑蘖 詘和搀 詘 这两种业务。但当前的版本中仅实现了撑蘖 詘业务，且在此版本的正式发行版中，用户（上层模块）不应产生搀 詘业务的请求，那么在测试时可用断言检查用户是否使用搀 詘业务。如下。

```
#define  EXAM_CONNECTIONLESS    0     // 无连接业务
#define  EXAM_CONNECTION        1     // 连接业务

int MsgProcess( _EXAM_MESSAGE *msg )
{
```

```
    unsigned char cService ;  /* 消息服务类 */

    EXAM_ASSERT( msg != NULL ) ;

cService = GetMsgServiceClass( msg ) ;

    EXAM_ASSERT( service != EXAM_CONNECTION ) ;  // 假设不使用连接业务

    ...  // 其它程序代码
}
```

<规则 10> 用断言对程序开发环境（OS/Compiler/Hardware）的假设进行检查。
程序运行时所需的软硬件环境及配置要求，不能用断言来检查，而必须由一段专门代码处理。用断言仅可对程序开发环境中的假设及所配置的某版本软硬件是否具有某种功能的假设进行检查。如某网卡是否在系统运行环境中配置了，应由程序中正式代码来检查；而此网卡是否具有某设想的功能，则可由断言来检查。
对编译器提供的功能及特性假设可用断言检查，原因是软件最终产品（即运行代码或机器码）与编译器已没有任何直接关系，即软件运行过程中（注意不是编译过程中）不会也不应该对编译器的功能提出任何需求。
示例：用断言检查编译器的 int 型数据占用的内存空间是否为 2，如下。

```
EXAM_ASSERT( sizeof( int ) == 2 ) ;
```

<规则 11> 正式软件产品中应把断言及其它调测代码去掉（即把有关的调测开关关掉）。

<规则 12> 用调测开关来切换软件的 DEBUG 版和正式版，而不要同时存在正式版本和 DEBUG 版本的不同源文件，以减少维护的难度。

<规则 13> 在软件系统中设置与取消有关测试手段，不能对软件实现的功能等产生影响。
即有测试代码的软件和关掉测试代码的软件，在功能行为上应一致。

<规则 14> 发现错误应该立即修改，并且若有必要记录下来。

<规则 15> 开发人员应坚持对代码进行彻底的测试（单元测试），而不依靠他人或测试组来发现问题。
<规则 16> 清理、整理或优化后的代码要经过审查及测试。
<规则 17> 代码版本升级要经过严格测试。

2.7. 代码编译

<规则 1> 打开编译器的所有告警开关对程序进行编译。

防止隐藏可能是错误的告警。

<规则 2> 在同一项目组或产品组中，要统一编译开关选项。

<规则 3> 某些语句经编译后产生告警，但如果你认为它是正确的，那么应通过某种手段去掉告警信息。

在 Borland C/C++中，可用“#pragma warn 鼙垂氐艋虼蚩 承 └婢 _

示例：

```
#pragma warn -rvl    // 关闭告警
int DoExample( void )
{
    // 程序，但无 return 语句。
}
#pragma warn +rvl    // 打开告警
```