# 代码规范V1.0

随着业务的扩展，策立公司的软件项目也转为多人协作开发模式，为了增加代码的可读性，需要制定统一的代码规范。本文档借鉴了一些其他公司常用的代码规范，结合我公司的实际情况编制而成。希望各位软件开发人员在以后的软件开发过程中能依照本代码规范来执行。

## 命名规则

代码中的变量名,函数数,类名不仅应该是是个有意义的词或短语,而且应该确切的表达了该变量或函数的功能.命名都应该用英文或者英文缩写,原则上不允许使用拼音或者拼音缩写。

命名方法我们参照匈牙利命名法和驼峰命名法。

匈牙利命名法的基本变现形式： 变量名=属性+类型+对象描述.

例如int m\_nNumber;

如果是局部变量一般就不需要属性了,所以就是int nNumber;

**1.其中对象描述的名称都要求有明确含义**，可以取对象名字全称或名字的一部分.可以起自解释的作用,从名字中就可以看出变量的功能.

**2.其中属性(有个下划线)包括**: g\_表示全局变量,  c\_表示常量 , m\_表示C++类成员变量, s\_ 表示静态变量 , sm\_表示静态成员变量

**3.类型部分都是用类型关键字的缩写表示**:

匈牙利命名法对于每种变量类型都有不同的缩写，我们在实际应中做了修改，如下：

|  |  |  |
| --- | --- | --- |
| 类型 | 简写 | 备注 |
| Float/doulbe | f |  |
| Int/long/short | n | 整形数据都可以用 |
| CString | str |  |
| BOOL/bool | b |  |
| 指针 | p |  |
| Char | c |  |
| DWORD | dw |  |
| HANDLE | h |  |
| … | … |  |

据说这种命名法是一们叫Charles Simonyi的匈牙利程序员发明的,他也在微软工作过.所以你查看MFC中的一些源码会发现都用的匈牙利命名法,你要是开发中用到MFC可能也倾向于使用这种命名规范.

匈牙利命名的优点显而易见,能从名字本身获取到很多信息,知道变量的类型或要实现的功能.

缺点是添加这些前缀会使变量名变长,变量名一长就会显得不雅,另外就是开发人员要多敲几个字母.当然还有其他很多缺点.

另外就是匈牙利命名主要针对基本类型变量名,但在面向对象语言中肯定到处是类名和函数名,那类名和函数名该怎么取呢? 此时在它们前面加个啥前缀没有太多意义的.

此时变量名=属性+类型+对象描述 ,中最有对象描述这一项可派得上用场了,但是对象描述可没给我们带来啥明确的格式指导,比如描述信息一长,就很不利于我们眼睛的识别.

**驼峰命名法**

骆驼有一个很明显的特征就是背像个山峰一样,有凹下去与凸起来的部分.

于是这种命名法就借鉴了这个特征,让变量名通过某些字母大写来达到凹凸的效果,这有利于眼睛识别,因为如果都是小写或大写都不利于眼睛识别.

**小驼峰命名法**

除第一个单词首字母小写,其他单词首字母大写.

例如 int myPhoneNumber;

**大驼峰命名法(又叫Pascal法)**

所以单词首字母大写

例如 void GetPhoneNumber();

**一般函数名和类名都用大驼峰命名法,不过在MFC中类名一般推荐是匈牙利与驼峰结合的变体,类前面加一个大写的字母C,后面部分就是大驼峰法了**

常见类型命名规范如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 命名方法 | 示例 | 备注 |
| 类名 | 驼峰命名法 | Class UIGrap  Class DataRecord |  |
| 方法和函数名 | 驼峰命名法 | Int GetMax()  Void SetValue(int nValue) |  |
| 变量名 | 匈牙利命名法 | 成员变量 int m\_nFactoryCount  局部变量 int nCurTemp  全局变量 int g\_nTextCount  常量 const Cstring c\_strFactoryName  ….. |  |
| 指针命名 | 匈牙利命名法 | Int \* pFactoryPos |  |
| 结构体或者联合体命名 | / | 1. 全部大写命名   Struct FACTORY   1. 匈牙利命名法   Struct Factory | 两种方法都可以 |
| 宏定义命名 | / | 全部大写  #define int INTNUM |  |
| C函数命名 | GUN规范 | 所有字母都使用小写，并用下划线 '\_' 作为单词的分隔。  int some \_function() |  |

## 2.    排版规则

**<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->)，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，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))

**<规则11>两个相邻的方法用空行隔开。**

Void Fun1()

{

}

Void Fun2()

{

}

## 3.    文档及注释

　　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。此规则非必要。**

　　例如:

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

　　文件名:

　　编写者:

　　编写日期:

　　简要描述:

　　修改记录:

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

**<规则5> 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6> 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7> 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](http://product.yesky.com/catalog/1467/" \t "_blank) \*/

　　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 8> 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。**

## 4.    编码要求

**<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](http://soft.yesky.com/security/)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，

示例:

　　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](http://product.yesky.com/catalog/215/" \t "_blank)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。**

**主要指大端模式和小端模式**

　　比如Intel CPU与68360 CPU，在处理位域及整数时，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 ... (字节的各撐粩)

　　\_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 ... (字节的各撐粩)

　　\_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](http://os.yesky.com/)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。**

　　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 ; // 此位域跨越字节摻唤訑处。

　　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nBackSum = nSum 没必要置于循环中

　　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。轮询判断除外。**

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

　　示例如下:

　　假设头文件为揇EF.INC"，则其内容应为:

　　#ifndef \_\_DEF\_INC

　　#define \_\_DEF\_INC

　　...

　　#endif

**<规则24> 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25> 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26> 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27> 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28> 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29> 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30> 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[]);

**<规则31> 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32> 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33> 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34> 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 ) ; //

　　... // 其它程序代码

　　}