# 实验一

# **C++**软件编码规范

目录

**[1.排版风格](#_Toc28769_WPSOffice_Level1)** **[1](#_Toc28769_WPSOffice_Level1)**

**[2. 可理解性](#_Toc13787_WPSOffice_Level1)** **[6](#_Toc13787_WPSOffice_Level1)**

[1.1 注释](#_Toc8310_WPSOffice_Level2) [6](#_Toc8310_WPSOffice_Level2)

[1.2 命名](#_Toc31999_WPSOffice_Level2) [11](#_Toc31999_WPSOffice_Level2)

[1.3 可维护性](#_Toc13243_WPSOffice_Level2) [12](#_Toc13243_WPSOffice_Level2)

[1.4. 程序正确性、效率](#_Toc14705_WPSOffice_Level2) [13](#_Toc14705_WPSOffice_Level2)

[1.5  接口](#_Toc15807_WPSOffice_Level2) [20](#_Toc15807_WPSOffice_Level2)

[1.6 代码可测性](#_Toc30455_WPSOffice_Level2) [22](#_Toc30455_WPSOffice_Level2)

**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->)，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)

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，憑捄蛻}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. 可理解性**

**1.1 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:摷蛞枋鰯一项描述本文件的目的和功能等。撔薷募锹紨是修改日志列表，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](http://product.yesky.com/catalog/1467/" \t "https://www.cnblogs.com/samuelwnb/p/_blank) \*/

　　AT(CHANNEL, "ATD\r",-1); /\*发送CNG ，接收 CED 和 HDLC 标志\*/

**1.2 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

　　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，即函数名应类似摱鼋峁箶。命名举例:

　　void PerformSelfTest(void) ;

　　void ProcChanAct(MSG\_CHAN\_ACTIV \*pMsg, UC MsgLen);

**1.3 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。

**1.4. 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](http://soft.yesky.com/security/" \t "https://www.cnblogs.com/samuelwnb/p/_blank)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“=斢霌==敗|斢霌||敗&斢霌&&數龋羝葱创砹耍嘁肫鞑灰欢芄患觳槌隼础\_

　　示例:如把“&斝闯蓳&&敚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](http://product.yesky.com/catalog/215/" \t "https://www.cnblogs.com/samuelwnb/p/_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。

　　比如Intel CPU与68360 CPU，在处理位域及整数时，其在内存存放的撍承驍，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/" \t "https://www.cnblogs.com/samuelwnb/p/_blank)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跀或撟謹的边界。

　　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 \*/

　　}

　　语句搉BackSum = nSum ;斖耆梢苑旁趂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“++敚瑩--敳僮鞔鎿+=1敚瑩-=1敚岣叱绦蛩俣取\_

　　<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摂，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。

**1.5  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:

　　假设头文件为揇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[]);

**1.6 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](http://product.yesky.com/catalog/821/" \t "https://www.cnblogs.com/samuelwnb/p/_blank)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 // 各模块共用的头文件

　　#include // 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 // 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 ) ; // 用断言检查摷偕柚刚氩晃諗这个条件

　　... // 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，准备提供撐蘖訑和摿訑 这两种业务。但当前的版本中仅实现了撐蘖訑业务，且在此版本的正式发行版中，用户(上层模块)不应产生摿訑业务的请求，那么在测试时可用断言检查用户是否使用摿訑业务。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。

　　<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

　　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