Tencent 腾讯

腾讯集团管理标准

GL/YF 014-2007V1.0-L1

C++编码规范

2007-10-25 发布　　2007-10-25 实施

腾讯集团 发布

前 言

本标准系公司首次发布实施，主要针对公司所有软件产品源代码范围的 C 和 C++编码风格，对 C 和 C++文件的版式、注释、标识符命名、可读性、变量、结构、函数和过程等方面均作出规范，以保障公司项目代码的易维护性和编码安全性。

本标准由研发管理部、即时通信产品部共同制定。

本标准主要起草人：Junjun(张莉珺)、qingming(王清明)、tracy(周银燕)

本标准主要审核人：anwenfeng(冯文信)、 paulinesong(宋虹漫)、ericlin（林松)、stevezheng(郑全战) 、echouzhou(周立)、polo(陈广域)、leon（郭凯天）

本标准批准人：jeffxiong(熊明华) 、charles(陈一丹)、tony（张志东)、ponyma（马化腾）

本标准首次发布日期：2007 年 10 月 25 日

本标准发送部门：公司各部门

C++编码规范

1 目的

为形成公司统一的C++编码风格，以保障公司项目代码的易维护性和编码安全性，特制定本规范。

2 适用范围

本标准适用于腾讯集团（含分公司等各级分支机构）所有使用C和C++作为开发语言的软件产品。

本标准中“腾讯集团”是指腾讯控股有限公司、其附属公司、及为会计而综合入账的公司，包括但不限于腾讯控股有限公司、深圳市腾讯计算机系统有限公司、腾讯科技（深圳）有限公司、腾讯科技（北京）有限公司、深圳市世纪凯旋科技有限公司、时代朝阳科技（深圳）有限公司、腾讯数码（深圳）有限公司、深圳市财付通科技有限公司。

3 总体原则

所有使用C和C++作为开发语言的软件产品都须遵照本规范的内容进行编码。

4 程序的版式

4.1 规则：程序块要采用缩进风格编写，缩进的空格数为 4 个。

说明：

由开发工具自动生成的代码可能不一致，但如果开发工具可以配置，则应该统一配置缩进为 4 个空格。

4.2 规则：缩进或者对齐只能使用空格键，不可使用 TAB 键。

说明 ：

使用 TAB 键需要设置 TAB 键的空格数目是 4 格。

4.3 规则：相对独立的程序块之间、变量说明之后必须加空行。

说明 ：

以下情况应该是用空行分开：

1）函数之间应该用空行分开；

2）变量声明应尽可能靠近第一次使用处，避免一次性声明一组没有马上使用的变量；

3）用空行将代码按照逻辑片断划分；

4）每个类声明之后应该加入空格同其他代码分开。

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;
```

4.4 规则：较长的语句（>80 字符）要分成多行书写。

说明：

以下情况应分多行书写：

1）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。

2）若函数或过程中的参数较长，则要进行适当的划分。

3）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：

```
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));

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 );

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
}
```

4.5 规则：不允许把多个短语句写在一行中，即一行只写一条语句。

说明：

一行代码只做一件事情，如只定义一个变量，或只写一条语句。这样的代码容易阅读，并且方便于写注释。

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;
```

4.6 规则：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;
}
```

4.7 规则：代码行之内应该留有适当的空格

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。代码行内应该适当的使用空格， 具体如下：

1）关键字之后要留空格。象 const、virtual、inline、case 等关键字之后至少要留一个空格， 否则无法辨析关键字。象 if、for、while 等关键字之后应留一个空格再跟左括号‘(’， 以突出关键字。

2）函数名之后不要留空格， 紧跟左括号’(‘， 以与关键字区别。

3）‘(’ 向后紧跟，‘)’、‘ ,’、‘ ;’ 向前紧跟， 紧跟处不留空格。

4）‘ ,’ 之后要留空格， 如 Function(x, y, z)。如果‘ ;’ 不是一行的结束符号， 其

后也要留空格， 如 for (initialization; condition; update)。

5）值操作符、比较操作符、算术操作符、逻辑操作符、位域操作符，如“ =”、“ +=” “ >=”、“ <=”、“ +”、“ *”、“ %”、“ &&”、“ ||”、“ <<” 、“ ^” 等二元操作符的前后应当加空格。

6）一元操作符如“ !”、“ ~”、“ ++”、“ --”、“ &”（ 地址运算符） 等前后不加空格。

7）象“[ ]”、“ .”、“ ->” 这类操作符前后不加空格。

示例：

```
应该按照以下的格式书写：
void foo()
{
      …// program code
}

if (i == 0)
{
      …// program code
}

foo->bar, foo.bar, foo[bar]

i++, !i, &i

i += 9, a *b
```

4.8 建议：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
}
```

5 注释

5.1 规则：源文件头部应进行注释，列出：生成日期、作者、模块目的/功能等。

示例：

下面这段源文件的头注释比较标准，可以不局限于此格式，但上述信息要包含在内。

```
/*************************************************************
  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。

也可以采用 javadoc 风格的文档注释，这里不再举例，下同。

5.2 规则：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、返回值等。

示例：

下面这段函数的注释比较标准，可以不局限于此格式，但上述信息要包含在内。

```
/*************************************************
  Description:    // 函数功能、性能等的描述
  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
*************************************************/
```

5.3 规则：注释应该和代码同时更新，不再有用的注释要删除。

5.4 规则：注释的内容要清楚、明了，不能有二义性。

说明：

错误的注释不但无益反而有害。

5.5 建议：避免在注释中使用非常用的缩写或者术语。

5.6 建议：注释的主要目的应该是解释为什么这么做，而不是正在做什么。如果从上下文不容易看出作者的目的，说明程序的可读性本身存在比较大的问题，应考虑对其重构。

5.7 建议：避免非必要的注释。

例如：

```
ClassA *pA = new ClassA();  //创建新实例
…
delete pA;  //销毁对象
```

5.8 规则：注释的版式

说明：注释也需要与代码一样整齐排版

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。

2）注释与所描述内容进行同样的缩排。

3）将注释与其上面的代码用空行隔开。

4）变量、常量、宏的注释应放在其上方相邻位置或右方。

示例：如下例子不符合规范。

```
例 1：注释在代码行之下
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
/* get replicate sub system index and net indicator */
```

```
例 2：缩进不统一
void example_fun( void )
{
/* code one comments */
    CodeBlock One

        /* code two comments */
    CodeBlock Two
}
```

```
例 3：代码过于紧凑
/* code one comments */
program code one
/* code two comments */
program code two
```

5.9 规则：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。

示例：

```
以下都是允许的注释方式
    /* active statistic task number */
    #define MAX_ACT_TASK_NUMBER 1000

    #define MAX_ACT_TASK_NUMBER 1000 /* active statistic task number */
```

5.10 规则：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可放在此域的右方。

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*/
    };
```

5.11 建议：对重要变量的定义需编写注释，特别是全局变量，更应有较详细的注释，包括对其功能、取值范围、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 － not in use         // 变量取值范围
 only  function  SCCPTranslate() in
 this module can modify it,  and  other
 module can visit it through call
 the  function GetGTTransErrorCode() */    // 使用方法
  BYTE g_GTTranErrorCode;
```

5.12 建议：分支语句（条件分支、循环语句等）需编写注释。

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。

5.13 规则：注释不宜过多，也不能太少，源程序中有效注释量控制在 20%~30%之间。

说明：

注释是对代码的“提示”，而不是文档，不可喧宾夺主，注释太多会让人眼花缭乱。

6 标识符命名

6.1 规则：命名尽量使用英文单词，力求简单清楚，避免使用引起误解的词汇和模糊的缩写，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  ;
```

6.2 规则：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。

说明：

1）如在 UNIX 系统，可采用全小写加下划线的风格或大小写混排的方式，但不能使用大小写与下划线混排的方式。

2）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。
```

6.3 建议：变量的命名可参考“匈牙利”标记法（Hungarian Notation）：TypePrefix+Name

例外：

C++程序不建议采用匈牙利命名法。因为 C++本身就是强类型语言，不需要像 C 一样用匈牙利命名法来强调变量类型。

有两个匈牙利命名法可以保留：m_xxxx 表示类的成员变量，g_xxx 表示全局变量。

说明：

以下的基本变量前缀可供参考：

前缀	含义	示例	说明
p	Pointer	char* pszName	很多情况下，p 总是和它所指向的变量的类型前缀一起使用。

s	String	String sName;	
str	CString	CString strName;	
sz psz	zero-terminated null-terminated string	char szName[16]; char* pszName;	
h	Handle	HWND hWindow	
c	Character (char)	char cLetter;	Sometimes *c* is used to denote a counter object.
by y	Byte or Unsigned Char	byte byMouthFull; byte yMouthFull;	
n	Integer (int)	int nSizeOfArray;	
f	Float	float fRootBeer;	
d	Double	double dDecker;	
b	Bool	bool bIsTrue; BOOL bIsTrue; int bIsTrue;	An integer can store a boolean value as long as you remember not to assign it a value other than 0 or 1
u	Unsigned...		
w	WORD		
l	Long	long lIdentifier;	Sometimes *l* is appended to *p* to denote that the pointer is a long. For example: lpszName is a long pointer to a zero-terminated string.
dw	DWORD		
C	Class	Class CObject; Class Object;	*C* is used heavily in Microsoft's Foundation Classes but using just a capital first letter is emphasized by Microsoft's J++.
S	Struct	struct SPlayer;	

I	Interface	class IMotion {
		public:
		virtual void Fly()
		= 0;
		};
			Used extensively in COM.

X	Nested Class	class CRocket {
		public:
		class
		XMotion:public
		IMotion {
		public:
		void Fly();
		} m_xUnknown;
		}
			Used extensively in COM.

x	Instantiation of a nested class.	class CAirplane {
		public:
		class
		XMotion:public
		IMotion {
		public:
		void Fly();
		} m_xUnknown;
		}
			Used extensively in COM.

m_	Class Member Identifiers	class CThing {
		private:
		int m_nSize;
		};	

g_	Global	String* g_psBuffer	Constant globals are usually in all caps. The *g_* would denote that a particular global is not a constant.
v	Void (no type)	void* pvObject	In most cases, *v* will be included with *p* because it is a common trick to typecast pointers to void pointers.
st	Struct variable	stPlayer	

6.4 规则：常量、宏和模板名采用全大写的方式， 每个单词间用下划线分隔。

6.5 建议：枚举类型 enum 常量应以大写字母开头或全部大写。

6.6 建议：命名中若使用了特殊约定或缩写，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。

6.7 规则：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）。

6.8 规则：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。

6.9 建议：除非必要，不要用数字或较奇怪的字符来定义标识符。

示例：

1）命名：

使人产生疑惑的命名：

```
#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 );
```

2）避免使用看上去相似的名称，如“l”、“1”和“I”看上去非常相似。

6.9 建议：函数名以大写字母开头，采用谓-宾结构（动-名），且应反映函数执行什么操作以及返回什么内容。

说明：

函数在表达式中使用，通常用于 if 子句，因此它们的意图应一目了然。

示例：

不好的命名：if (CheckSize(x))

没有帮助作用，因为它没有告诉我们 CheckSize 是在出错时返回 true 还是在不出错时返回 true。

好的命名：if (ValidSize(x))

则使函数的意图很明确。

6.10 建议：类、结构、联合、枚举的命名须分别以 C、S、U、E 开头，其他部分遵从一般变量命名规范。

7 可读性

7.1 规则：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。

7.2 建议：不要编写太复杂 、多用途的复合表达式。

7.3 规则：涉及物理状态或者含有物理意义的常量，避免直接使用数字，必须用有意义的枚举或常量来代替。

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。

```
const int TRUNK_IDLE = 0;
const int TRUNK_BUSY = 1;

if (Trunk[index].trunk_state == TRUNK_IDLE)
{
    Trunk[index].trunk_state = TRUNK_BUSY;
    ...  // program code
}
```

7.4 规则：禁止使用难以理解，容易产生歧义的语句。

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; ”
```

8 变量、结构

8.1 建议：尽量少使用全局变量，尽量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。

8.2 规则：变量，特别是指针变量，被创建之后应当及时把它们初始化，以防止把未被初始化的变量当成右值使用。

说明：在 C/C++中引用未经赋值的指针，经常会引起系统崩溃。

8.3 建议：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;
```

8.4 建议：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。

8.5 建议：尽量减少没有必要的数据类型默认转换与强制转换。

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。

8.6 规则：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：

1）在 Intel CPU 与 SPARC CPU，在处理位域及整数时，其在内存存放的“顺序”正好相反。

2）在对齐方式下，CPU 的运行效率要快一些。

示例：

a.字节顺序

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 SPARC 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）

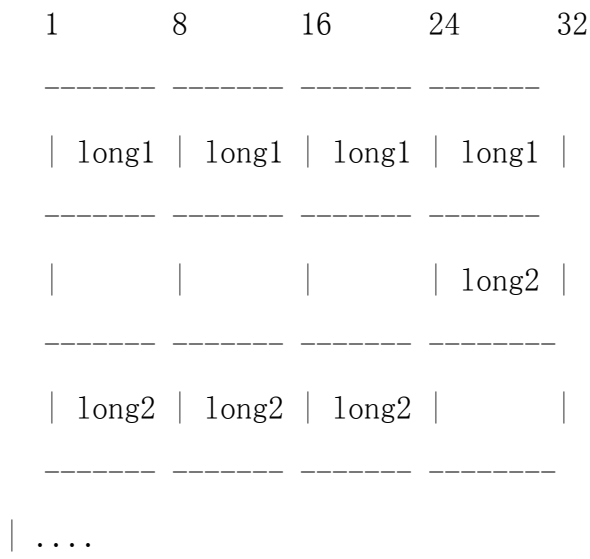
exam exam 高字节 exam 低字节

内存：　　7 bit　　6 bit　　5 bit　　...　（字节的各“位”）

EXAM_BIT　　A1　　A2　　A3

b.对齐

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。



| ....

9　函数、过程

9.1 规则：调用函数要检查所有可能的返回情况，不应该的返回情况要用 ASSERT 来确认。

9.2 建议：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。

9.3 建议：调用公共接口函数时，调用者有保障调用参数符合要求的义务。作为一种防

御性的编程风格，被调用函数也应该对传入参数做必要的安全检查。

9.4 建议：函数的规模尽量限制在 100 行以内。

说明：不包括注释和空格行。

9.5 建议：一个函数仅完成一件功能。

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。

9.6 建议：不能用 ASSERT 代替必要的安全处理代码，确保发布版的程序也能够合理地处理异常情况。

实例：

```
void Reset(int *p)
{
    assert(NULL != p);
    if(NULL != p)      //不能因为前面的 assert，省略此处的判断
    {
        *p = 0;
    }
}
```

9.7 尽量写类的构造、拷贝构造、析构和赋值函数 ，而不使用系统缺省的。

说明：

编译器以“位拷贝”的方式自动生成缺省的拷贝构造函数和赋值函数 ，倘若类中含有指针变量，那么这两个缺省的函数就隐含了错误。

示例：

假设有以下的类定义：

```
class String
{
    public:
        String(const char *str = NULL);  // 普通构造函数
        String(const String &other);     // 拷贝构造函数
        ~ String(void);                  // 析构函数
```

```
        String & operate =(const String &other);  // 赋值函数
    private:
        char     *m_data;                // 用于保存字符串
};
String  a , b ;
```

如果将 a 赋给 b，缺省赋值函数的“位拷贝”意味着执行 b.m_data = a.m_data。这将造成以下的错误：

1）b.m_data 原有的内存没被释放，造成内存泄露；

2）b.m_data 和 a.m_data 指向同一块内存，a 或 b 任何一方变动都会影响另一方；

3）在对象被析构时，m_data 被释放了两次。

9.8 建议：对于不需要拷贝构造函数时，应显式地禁止它，避免编译器生成默认的拷贝构造函数。

示例：

```
class CObject
{
public:
    CObject（）;
private:
    CObject & CObject(const CObject rhv) ; //定义但不实现

}
```

9.9 建议：谨慎使用与程序运行的环境相关的系统函数。

示例：如 strxfrm （） 和 strcoll （） ，这两个函数依赖于 LC_COLLATE 的设置。如果进程所运行的环境变量没有与开发环境一样设置，可能会产生错误的结果。

9.10 建议：禁止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
```

9.11 规则：检查函数所有参数与非参数的有效性。

说明：

1）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。

2）不应该的入口情况要用 ASSERT 来确认。

3）有时候不处理也是一种处理，但要明确哪些情况不处理。try...catch 是一种常用的不处理的处理手段。

9.12 建议：函数实现中不改变内容的参数要定义成 const。

示例：

```
int GetStrLen(const char*);
int GetNumberCount(const CString&);
```

9.13 规则：函数的返回值要清楚、明了，让使用者不容易忽视错误情况。

说明：

函数的每种出错返回值的意义要清晰、明了、准确，防止使用者误用、理解错误或忽视错误返回码。

10 C++专用规范

10.1 规则：在高警告级别下干净地编译。

使用编译器的最高警告级别。要求干净的（没有警告的）构建（build）并理解所有的警告。通过修改代码来消除警告，而不是通过降低警告级别来消除。对于明确理解其含义，确信不会造成任何问题的警告，则可以局部关闭。

10.2 规则：确保资源为对象所占有，使用显式的 RAII 和智能指针。

C++在语言层面强制的构造/析构恰好与资源获取/释放这对函数相对应，在处理需要调用成对的获取/释放函数的资源时，应将该资源封装在对象中，并在对象的析构函数中释放该资源，这样就保证了获取/释放的匹配。

最好用智能指针来保存动态分配的资源，而不要用原始指针。

10.3 规则：主动使用 const，避免使用宏。

应该尽可能的使用常量而不用变量，另外在定义数值的时候，应该把 const 做为默认的选项。它是安全的，在编译的时候（参见附录 C《编码安全规范》）检查，它集成在 C++的类型系统中。除非要调用一个非 const 函数，否则不要强制去除 const。

宏无视作用域，无视类型系统，无视所有其它的语言特性和规则，并从#define 处开始将该符号劫持。只有对少数的重要任务，宏仍是仅有的解决方案，如#include 防护哨，用于条件编译的#ifdef 和#if defined，以及用来实现 assert。

10.4 规则：合理使用组合(composition)和继承(inheritance)。

继承是 C++中耦合度最强的关系之一。软件工程的一条重要原则是尽量减少耦合，在组合和继承都能均可适用的情况下，应该优先考虑使用组合。组合的意思是将一种类型以成员变量方式嵌入相关类型中。组合有如下优点：

1）在不影响调用代码的同时也更灵活。

2）编译期绝缘性好，编译时间也能缩短。

3）代码不可预测程度降低（有些类不适合作为基类）。

10.5 规则：尽可能局部地声明变量。

尽可能局部地声明每个变量，这通常是在程序具备了足够的数据来初始化变量之后，并紧接着首次使用该变量之前。

例外：

1）有时将变量从循环内提出到循环外是有益的。

2）由于常量不增加状态，因此本条对常量不适用。

10.6 规则：通过值，（智能）指针，或引用适当地取得参数。

对仅用于输入的参数来说：

1）始终给仅用于输入的指针或引用参数加上 const 限定符。

2）最好是通过原始类型（例如：char，float）和可以通过值来复制并且复制成本低的值对象（例如：Point，complex<float>）来取得参数。

3）对其它自定义类型的输入，最好是通过 const 引用来取得。

4）如果函数需要参数的复本，那么可以考虑用传递值来代替传递引用。从概念上说，这等价于取得一个 const 引用再做一次复制，它可以帮助编译器更好地优化掉临时对象。

对输出或输入/输出参数来说：

1）如果参数是可选的（因此调用方可以传递空指针来表示“不可用”或“不关心”的值），或者函数要保存指针的一个复本或操控参数的所有权，那么最好是通过（智能）指针传递。

2）如果参数是必需的，而且函数无需保存指向该参数的指针或无需操控参数的所有权，那么最好是通过引用传递。这表明该参数是必需的，并让调用方来负责提供一个有效的对象。

10.7 规则：不要在头文件中定义具有链接属性的实体。

重复导致膨胀：

具有链接属性的实体，包括名字空间层级的变量或函数，需要占用内存。把此类实体定义在头文件中会导致编译错误或内存浪费。应该把具有链接属性的实体放在实现文件中。

下面这些具有外部链接属性的实体可以放在头文件中：

1）内联函数：虽然它们具有外部链接属性，但是链接器会保证不拒绝链接多个复本。除此之外，它们的行为和普通的函数完全一样。

2）函数模板（Function templates）：与内联函数相似，除了重复的复本是可接受的之外（最好是完全一样的），模板实例化的行为与普通的函数一样。而一个好的编译系统会消除无用的复本。

3）类模板的静态数据成员：这对链接器来说可能有点粗暴，不过只需在头文件中定义它们，则可让编译器和链接器来处理剩余的事情。

10.8 规则：尽量用异常来报告错误。

与错误码相比，要尽量用异常来报告错误。对一些无法使用异常的错误，或者一些不属于错误的情况，可以用状态码（status code，例如：返回码，errno）来报告。如果不可能或不需要从错误中恢复，那么可以使用其它方法，比如正常或非正常地终止程序。

在C++中，和用错误码来报告错误相比，用异常来报告错误具有许多明显的优势，所有这些都使得编出来的代码更健壮：

1）程序员不能无视异常：错误码的最糟糕的缺点就是在默认情况下它们会被忽略；即使是给予错误码微不足道的关注，都必须显式地编写代码，以接受错误并做出反应。程序员因为偶然（或因为懒惰）而忘记关注错误码是很平常的事。这使得代码复查变得更困难。程序员不能无视异常；要忽略异常，必须显式地捕获它（即使只是用catch(...)），然后不对之进行处理。

2）异常会自动传递：默认情况下错误码不会跨作用域传递；为了把一个低层的错误码通知高层的调用函数，程序员必须在中间层的代码中显式地手工编写代码以传递该错误。异常会自动地跨作用域传递，直到被处理为止。（“试图使每个函数都成为防火墙并不是一种好办法。”[Stroustrup94, §16.8] ）

3）异常处理从主控制流中去除了错误处理及恢复：错误码的检测及处理，一旦要写的话，就必须夹杂在主控制流中（并使之变得难以理解）。这使得主控制流以及错误处理的代码都更难以理解和维护。异常处理很自然地把错误检测及恢复移到醒目的catch代码块中，即它使错误处理既醒目，又易于使用，而不是纠缠在主控制流中。

11 附则

本规范由研发管理部、即时通信产品负责修订、解释，本规范自发布之日起实施。

附录A
（规范性附录）
程序效率

1 规则：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：

1）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）不能一味地为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。

2 规则：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。

3 建议：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。应作如下改进（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
{
    BYTE name[8];
    BYTE age;
    BYTE sex;
    BYTE class;
} STUDENT;

typedef struct STUDENT_SCORE_STRU
{
    WORD student_index;
    BYTE subject;
    float score;
} STUDENT_SCORE;
```

4 规则：循环体内工作量最小化

说明：

1）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。

2）在多重循环中，应将最忙的循环放在最内层，这样可以减少 CPU 切入循环层的次数，从而提高效率。

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 */
```

5 建议：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。

6 建议：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，故可作如下改进，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;
    }
}
```

7 建议：在逻辑清楚且不影响可读性的情况下，代码越少越好。

说明：

写程序不是以行数的多少来判断一个人的工作效率，代码不是越多越好。

8 规则：尽量使用标准库函数，不要“发明”已经存在的库函数。

9 建议：要尽量重用已有的代码，直接调用已有的 API 。

说明：

如果原有的代码质量比较好，尽量复用它。但是不要修补很差劲的代码，应当重新编写。

附录B
（规范性附录）
质量保证

1 规则：只引用属于自己的存贮空间。

说明：

若模块封装得较好，那么一般不会发生非法引用他人的空间的情况。

2 规则：防止引用已经释放的内存空间。

说明：

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 C 语言指针），而另一模块在随后的某个时刻又使用了它。

3 规则：过程/函数中动态分配的资源（包括内存、文件等），在过程/函数退出之前要释放。

示例：

以下例子将造成内存泄露 ：

```
void Func(void)
{
    char *p = (char *) malloc(128);
    …… //do something
    return ;
}
```

4 建议：充分理解 new/delete，malloc/free 等指针相关的函数的意义，对指针操作时需小心翼翼。

示例：

1）内存被释放了，并不表示指针会消亡或者成了 NULL 指针。

```
char *p = (char *) malloc(100);
strcpy(p, “hello”);
free(p);
…… //do something

if (p != NULL)
{
    …… //These code will be executed ?
}
```

p 没有成为 NULL 指针，if 中间的代码被错误执行了。

2）new 与 delete 要匹配。

错误的写法：

```
void Func()
{
    Object *pObjects=new Object[10] ;
    …… // do something
    delete pObjects ;
}
```

正确的写法：

```
void Func()
{
    Object *pObjects=new Object[10] ;
    …… // do something
    delete  []pObjects ;
}
```

5 规则：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。

示例：

假设某软件系统最多可由 10 个用户同时使用，用户号为 1-10，那么如下程序存在问题。

```
#define MAX_USR_NUM 10

unsigned char usr_login_flg[MAX_USR_NUM]= "";


void set_usr_login_flg( unsigned char usr_no )

{

    if (!usr_login_flg[usr_no])
```

```
    {
        usr_login_flg[usr_no]= TRUE;
    }
}
```

当 usr_no 为 10 时，将使用 usr_login_flg 越界。可采用如下方式解决：

```
void set_usr_login_flg( unsigned char usr_no )
{
    if (!usr_login_flg[usr_no - 1])
    {
        usr_login_flg[usr_no - 1]= TRUE;
    }
}
```

6 建议：要时刻注意易混淆的操作符。当编完程序后，应从头至尾检查一遍这些操作符，以防止拼写错误。

说明：

形式相近的操作符最容易引起误用，如 C/C++中的“=”与“==”、“|”与“||”、“&”与“&&”等，若拼写错了，编译器不一定能够检查出来。

示例：

如把“&”写成“&&”，或反之。

```
ret_flg = (pmsg->ret_flg & RETURN_MASK);
```

被写为：

```
ret_flg = (pmsg->ret_flg && RETURN_MASK);
```

```
rpt_flg = (VALID_TASK_NO( taskno ) && DATA_NOT_ZERO( stat_data ));
```

被写为：

```
rpt_flg = (VALID_TASK_NO( taskno ) & DATA_NOT_ZERO( stat_data ));
```

7 建议：条件表达式要把常量写在前面。

说明：

习惯写 if (MAX_COUNT == nIndex) 就不会发生 if (nIndex = MAX_COUNT)的错误。

8 建议：有可能的话，if 语句尽量加上 else 分支，对没有 else 分支的语句要小心对待；switch 语句必须有 default 分支。

9 规则：尽量少用 goto 语句。

说明：

1）goto 语句会破坏程序的结构性，所以除非确实需要，最好不使用 goto 语句。

2）使用 goto 语句时，不能往回跳。

3）尽量不要用多于一个的 goto 语句标记。

10 规则：不使用与硬件、操作系统、或编译器相关的语句，而使用建议的标准语句，以提高软件的可移植性和可重用性。

示例：

```
for( idx=0 ; idx<40; dest[idx]=src[idx++] )  ;
```

这段代码中，在不同的操作系统 ，有不同的执行顺序。是先执行 idx++ 呢？还是先执行 dest[idx]=src[idx++] ？

正确的写法是：

```
for (idx=0 ; idx<40; idx ++)
{
   dest[idx]=src[idx] ;
}
```

11 建议：时刻注意表达式是否会上溢、下溢。

示例：如下程序将造成变量下溢。

```
unsigned char size ;
while (size-- >= 0) // 将出现下溢
```

```
{
    ... // program code
}
```

当 size 等于 0 时，再减 1 不会小于 0，而是 0xFF，故程序是一个死循环。应如下修改。

```
char size; // 从 unsigned char 改为 char
while (size-- >= 0)
{
    ... // program code
}
```

12 规则：使用第三方提供的软件开发工具包或控件时，要注意以下几点：

1）充分了解应用接口、使用环境及使用时注意事项。

2）不能过分相信其正确性。

3）除非必要，不要使用不熟悉的第三方工具包与控件。

说明：

使用工具包与控件，可加快程序开发速度，节省时间，但使用之前一定对它有较充分的了解，同时第三方工具包与控件也有可能存在问题。

13 规则：资源文件（多语言版本支持），如果资源是对语言敏感的，应让该资源与源代码文件脱离，具体方法有下面几种：使用单独的资源文件、DLL 文件或其它单独的描述文件（如数据库格式）。

14 规则：打开编译器的所有告警开关对程序进行编译，并且要确认、处理所有的编译告警。

15 建议：通过代码走读及审查方式对代码进行检查。

说明：

代码走读主要是对程序的编程风格如注释、命名等以及编程时易出错的内容进行检查，可由开发人员自己或开发人员交叉的方式进行；代码审查主要是对程序实现的功能及程序的稳定性、安全性、可靠性等进行检查及评审，可通过自审、交叉审核或指定部门抽查等方式进行。

16 建议：如果可能，单元测试要覆盖 98%以上的代码，尽可能早地发现和解决问题。

说明：

1）尽早发现问题可以避免问题的扩大化，减少运维成本。

2）开发人员要树立“不要依赖测试人员”的观念，且不要抱侥幸心理，会出问题的地方总是会出问题的。

17 建议：如果可能，尽量使用 pc-lint，purify，LogiScope 等测试工具，以提高效率。

附录C
（规范性附录）
编码安全规范

编码安全规范v1.0.doc