Q/ZX

深圳市中兴通讯股份有限公司企业标准

（设计技术标准）

Q/ZX 04.302.1–2003

软件编程规范 — C/C++

2003-01-04 发布　　2003-01-06 实施

深圳市中兴通讯股份有限公司　发布

目　　次

前　言

编写本标准的目的是为了统一公司软件编程风格，提高软件源程序的可读性、可靠性和可重用性，提高软件源程序的质量和可维护性，减少软件维护成本，最终提高软件产品生产力。

本规范是针对 C/C++语言的编程规范，其它不同编程语言可以参照此规范执行。本规范适用于公司所有产品的软件源程序，同时考虑到不同产品和项目的实际开发特性，本规范分成规则性和建议性两种：对于规则性规范，要求所有软件开发人员严格执行；对于建议性规范，各项目编程人员可以根据实际情况选择执行。本规范的示例都以 C/C++语言描述。

本规范的内容包括：基本原则、布局、注释、命名规则、变量常量与类型、表达式与语句、函数与过程、可靠性、可测性、断言与错误处理等。规范最后给出了标准模版供软件人员参考。

本规范由软件编程规范 C/C++小组编写，主要成员如下：

技术中心研究部：李军、刘继兴

技术中心成都所：左雪梅

本部事业部：李晖

网络事业部：田小渝、许生海、徐火顺、黄志强

CDMA 事业部：程远忠、吴应祥

移动事业部：吴从海、王宏伟

软件编程规范有系列标准，包括：C/C++规范、GUI 规范、Delphi 规范、Java 规范。

本标准的附录 A、附录 B 是资料性附录。

自本标准实施之日起，以后新编写的和修改的代码均应执行本标准。

本标准由深圳市中兴通讯股份有限公司技术中心研究部提出，技术中心技术管理部归口。

本标准起草部门：技术中心研究部。

本标准主要起草人：软件编程规范小组。

本标准于 2003 年 1 月首次发布。

软件编程规范 — C/C++

1 范围

本标准规定了C/C++语言的编程规范。

本标准适用于公司内使用C/C++语言编码的所有软件。本规范自生效之日起，对以后新编写的和修改的代码有约束力。

2 术语和定义

下列术语和定义适用于本标准。

2.1 原则

编程时应该坚持的指导思想。

2.2 规则

编程时必须遵守的约定。

2.3 建议

编程时必须加以考虑的约定。

2.4 说明

对此规则或建议的必要的解释。

2.5 正例

对此规则或建议给出的正确例子。

2.6 反例

对此规则或建议给出的反面例子。

3 基本原则

【原则 1-1】首先是为人编写程序，其次才是计算机。

说明：这是软件开发的基本要点，软件的生命周期贯穿产品的开发、测试、生产、用户使用、版本升级和后期维护等长期过程，只有易读、易维护的软件代码才具有生命力。

【原则 1-2】保持代码的简明清晰，避免过分的编程技巧。

说明：简单是最美。保持代码的简单化是软件工程化的基本要求。不要过分追求技巧，否则会降低程序的可读性。

【原则 1-3】所有的代码尽量遵循 ANSI C 标准。

说明：所有的代码尽可能遵循 ANSI C 标准，尽可能不使用 ANSI C 未定义的或编译器扩展的功能。

【原则 1-4】编程时首先达到正确性，其次考虑效率。

说明：编程首先考虑的是满足正确性、健壮性、可维护性、可移植性等质量因素，最后才考虑程序的效率和资源占用。

【原则 1-5】避免或少用全局变量。

说明：过多地使用全局变量，会导致模块间的紧耦合，违反模块化的要求。

【原则 1-6】尽量避免使用 GOTO 语句。

【原则 1-7】尽可能重用、修正老的代码。

说明：尽量选择可借用的代码，对其修改优化以达到自身要求。

【原则 1-8】 尽量减少同样的错误出现的次数。

说明：事实上，我们无法做到完全消除错误，但通过不懈的努力，可以减少同样的错误出现的次数。

4 布局

程序布局的目的是显示出程序良好的逻辑结构，提高程序的准确性、连续性、可读性、可维护性。更重要的是，统一的程序布局和编程风格，有助于提高整个项目的开发质量，提高开发效率，降低开发成本。同时，对于普通程序员来说，养成良好的编程习惯有助于提高自己的编程水平，提高编程效率。因此，统一的、良好的程序布局和编程风格不仅仅是个人主观美学上的或是形式上的问题，而且会涉及到产品质量，涉及到个人编程能力的提高，必须引起大家重视。

4.1 文件布局

【规则 2-1-1】遵循统一的布局顺序来书写头文件。

说明：以下内容如果某些节不需要，可以忽略。但是其它节要保持该次序。

头文件布局:

文件头（参见第三章“注释”）
#ifndef 文件名_H（全大写）
#define 文件名_H
其它条件编译选项
#include（依次为标准库头文件、非标准库头文件）
常量定义
全局宏
全局数据类型
类定义
模板（template）（包括C++中的类模板和函数模板）
全局函数原型
#endif

【规则 2-1-2】遵循统一的布局顺序来书写实现文件。

说明：以下内容如果某些节不需要，可以忽略。但是其它节要保持该次序。

实现文件布局：

文件头（参见第三章“注释”）
#include（依次为标准库头文件、非标准库头文件）
常量定义
文件内部使用的宏
文件内部使用的数据类型
全局变量
本地变量（即静态全局变量）
局部函数原型
类的实现
全局函数
局部函数

【规则 2-1-3】使用注释块分离上面定义的节。

正例：

```
/ ************************************************************
*                    数据类型定义                              *
************************************************************ /
typedef unsigned char BOOLEAN;
```

```
/*****************************************************************
 *                          函数原型                             *
 *****************************************************************/
 int DoSomething(void);
```

【规则 2-1-4】头文件必须要避免重复包含。

说明：可以通过宏定义来避免重复包含。

正例：

```
#ifndef MODULE_H
#define MODULE_H

[文件体]

#endif
```

【规则 2-1-5】包含标准库头文件用尖括号 <>，包含非标准库头文件用双引号“”。

正例：

```
#include <stdio.h>
#include “heads.h”
```

【规则 2-1-6】遵循统一的顺序书写类的定义及实现。

说明：

类的定义（在定义文件中）按如下顺序书写：

公有属性
公有函数
保护属性
保护函数
私有属性
私有函数

类的实现（在实现文件中）按如下顺序书写：

构造函数
析构函数
公有函数
保护函数
私有函数

4.2 基本格式

【规则 2-2-1】程序中一行的代码和注释不能超过 80 列。

说明：包括空格在内不超过 80 列。

【规则 2-2-2】if、else、else if、for、while、do 等语句自占一行，执行语句不得紧跟其后。不论执行语句有多少都要加 { }。

说明：这样可以防止书写失误，也易于阅读。

正例：

```
if (varible1 < varible2)
{
    varible1 = varible2;
}
```

反例：下面的代码执行语句紧跟 if 的条件之后，而且没有加{}，违反规则。

```
if (varible1 < varible2) varible1 = varible2;
```

【规则 2-2-3】定义指针类型的变量，*应放在变量前。

正例：

```
float  *pfBuffer;
```

反例：

```
float*  pfBuffer;
```

〖建议 2-2-1〗源程序中关系较为紧密的代码应尽可能相邻。

说明：这样便于程序阅读和查找。

正例：

```
iLength    = 10;
iWidth     = 5;      // 矩形的长与宽关系较密切，放在一起。
StrCaption = “Test”;
```

反例：

```
iLength    = 10;
strCaption = “Test”;
iWidth     = 5;
```

4.3 对齐

【规则 2-3-1】 禁止使用 TAB 键，必须使用空格进行缩进。缩进为 4 个空格。

说明：消除不同编辑器对 TAB 处理的差异，有的代码编辑器可以设置用空格代替 TAB 键。

【规则 2-3-2】程序的分界符‘{’和‘}’应独占一行并且位于同一列，同时与引用它们的语句左对齐。{ }之内的代码块使用缩进规则对齐。

说明：这样使代码便于阅读，并且方便注释。

do while 语句和结构的类型化时可以例外，while 条件和结构名可与 } 在同一行。

正例：

```
void Function(int iVar)
{                        // 独占一行并与引用语句左对齐。
    while (condition)
    {
        DoSomething();   // 与{ }缩进 4 格
    }
}
```

反例：

```
void Function(int iVar){
    while (condition){
        DoSomething();
    }}
```

【规则 2-3-3】声明类的时候，public、protected、private 关键字与分界符{} 对齐，这些部分的内容要进行缩进。

正例：

```
class CCount
{
public:                          // 与 { 对齐
    CCount (void);               // 要进行缩进
    ~ CCount (void);
    int GetCount(void);
    void SetCount(int iCount);

private:
    int m_iCount;
}
```

【规则 2-3-4】结构型的数组、多维的数组如果在定义时初始化，按照数组的矩阵结构分行书写。

正例：

```
int aiNumbers[4][3] =
{
      1, 1, 1,
      2, 4, 8,
      3, 9, 27,
      4, 16, 64
}
```

【规则 2-3-5】相关的赋值语句等号对齐。

正例：

```
tPDBRes.wHead      =  0;
tPDBRes.wTail      =  wMaxNumOfPDB - 1;
tPDBRes.wFree      =  wMaxNumOfPDB;
tPDBRes.wAddress   =  wPDBAddr;
tPDBRes.wSize      =  wPDBSize;
```

〖建议 2-3-1〗在 switch 语句中，每一个 case 分支和 default 要用{ }括起来，{ }中的内容需要缩进。

说明：使程序可读性更好。

正例：

```
switch (iCode)
{
    case 1:
    {
        DoSomething();     // 缩进 4 格
        break;
    }
    case 2:
    {                      // 每一个 case 分支和 default 要用{}括起来
        DoOtherThing();
        break;
    }
    …                      // 其它 case 分支
```

```
        default:
        {
            DoNothing();
            break;
        }
    }
```

4.4 空行空格

【规则 2-4-1】不同逻辑程序块之间要使用空行分隔。

说明：空行起着分隔程序段落的作用。适当的空行可以使程序的布局更加清晰。

正例：

```
void Foo::Hey(void)
{
    [Hey 实现代码]
}
      // 空一行
void Foo::Ack(void)
{
    [Ack 实现代码]
}
```

反例：

```
void Foo::Hey(void)
{
    [Hey 实现代码]
}
void Foo::Ack(void)
{
    [Ack 实现代码]
}
// 两个函数的实现是两个逻辑程序块，应该用空行加以分隔。
```

【规则 2-4-2】一元操作符如“!”、“~”、“++”、“--”、“*”、“&”（地址运算符）等前后不加空格。“ [ ]”、“.”、“->”这类操作符前后不加空格。

正例：

```
!bValue
~iValue
++iCount
*strSource
```

```
&fSum
aiNumber[i] = 5;
tBox.dWidth
tBox->dWidth
```

【规则 2-4-3】多元运算符和它们的操作数之间至少需要一个空格。

正例：

```
fValue  =  fOldValue;
fTotal  +  fValue
iNumber +=  2;
```

【规则 2-4-4】关键字之后要留空格。

说明：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。

【规则 2-4-5】函数名之后不要留空格。

说明：函数名后紧跟左括号‘(’，以与关键字区别。

【规则 2-4-6】‘(’向后紧跟，‘)’、‘,’、‘;’向前紧跟，紧跟处不留空格。‘,’之后要留空格。‘;’不是行结束符号时其后要留空格。

正例：

例子中的 ␣ 代表空格。

```
for␣(i␣=␣0;␣i␣<␣MAX_BSC_NUM;␣i++)
{
    DoSomething(iWidth,␣iHeight);
}
```

【规则 2-4-7】注释符与注释内容之间要用一个空格进行分隔。

正例：

```
/* 注释内容 */
// 注释内容
```

反例：

```
/*注释内容*/
//注释内容
```

4.5 断行

【规则 2-5-1】长表达式（超过 80 列）要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐。

说明：条件表达式的续行在第一个条件处对齐。

for 循环语句的续行在初始化条件语句处对齐。

函数调用和函数声明的续行在第一个参数处对齐。

赋值语句的续行应在赋值号处对齐。

正例：

```
if ((iFormat == CH_A_Format_M)
  && (iOfficeType == CH_BSC_M)) // 条件表达式的续行在第一个条件处对齐
{
    DoSomething();
}

for (long_initialization_statement;
    long_condiction_statement;     // for 循环语句续行在初始化条件语句处对齐
    long_update_statement)
{
    DoSomething();
}

// 函数声明的续行在第一个参数处对齐
BYTE ReportStatusCheckPara(HWND hWnd,
                           BYTE ucCallNo,
                           BYTE ucStatusReportNo);

// 赋值语句的续行应在赋值号处对齐
fTotalBill = fTotalBill + faCustomerPurchases[iID]
          + fSalesTax(faCustomerPurchases[iID]);
```

【规则 2-5-2】函数声明时，类型与名称不允许分行书写。

正例：

```
extern double FAR CalcArea(double dWidth, double dHeight);
```

反例：

```
extern double FAR
CalcArea(double dWidth, double dHeight);
```

5 注　释

注释有助于理解代码，有效的注释是指在代码的功能、意图层次上进行注释，提供有用、额外的信息，而不是代码表面意义的简单重复。

【规则 3-1】C 语言的注释符为“/* … */”。C++语言中，多行注释采用“/* … */”，单行注释采用“// …”。

【规则 3-2】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，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。有效的注释是指在代码的功能、意图层次上进行注释，提供有用、额外的信息。

【规则 3-3】注释使用中文。

说明：对于特殊要求的可以使用英文注释，如工具不支持中文或国际化版本时。

【规则 3-4】文件头部必须进行注释，包括：.h 文件、.c 文件、.cpp 文件、.inc 文件、.def 文件、编译说明文件.cfg 等。

说明：注释必须列出：版权信息、文件标识、内容摘要、版本号、作者、完成日期、修改信息等。

正例：

下面是文件头部的中文注释：

```
/*********************************************************************
* 版权所有 (C)2001, 深圳市中兴通讯股份有限公司。
*
* 文件名称： // 文件名
* 文件标识： // 见配置管理计划书
* 内容摘要： // 简要描述本文件的内容，包括主要模块、函数及其功能的说明
* 其它说明： // 其它内容的说明
* 当前版本： // 输入当前版本
* 作　　者： // 输入作者名字及单位
* 完成日期： // 输入完成日期，例：2000年2月25日
*
* 修改记录1：// 修改历史记录，包括修改日期、修改者及修改内容
*　　修改日期：
```

```
*     版 本 号：
*     修 改 人：
*     修改内容：
* 修改记录2：...
**********************************************************************/
```

下面是文件头部的英文注释：

```
/**********************************************************************
* Copyright (C) 2001, ZTE Corporation.
*
* File Name:      // 文件名 （注释对齐）
* File Mark:      // 见配置管理计划书
* Description:    // 简要描述本文件的内容，完成的主要功能
* Others:         // 其它内容的说明
* Version:        // 输入当前版本
* Author:         // 输入作者名字及单位
* Date:           // 输入完成日期，例：2001-12-12
*
* History 1:           // 修改历史记录，包括修改日期、修改者及修改内容
*     Date:
*     Version:
*     Author:
*     Modification:
* History 2: ...
**********************************************************************/
```

【规则 3-5】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、返回值、访问和修改的表、修改信息等。

说明：注释必须列出：函数名称、功能描述、输入参数、输出参数、返 回 值、修改信息等。

正例：

下面是函数头部的中文注释：

```
/**********************************************************************
* 函数名称：  // 函数名称
* 功能描述：  // 函数功能、性能等的描述
* 访问的表：  //（可选）被访问的表，此项仅对于有数据库操作的程序
* 修改的表：  //（可选）被修改的表，此项仅对于有数据库操作的程序
* 输入参数：  // 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
* 输出参数：  // 对输出参数的说明。
* 返 回 值：  // 函数返回值的说明
```

```
* 其它说明：  // 其它说明
* 修改日期        版本号      修改人          修改内容
* -----------------------------------------------
* 2002/08/01      V1.0        XXXX            XXXX
*********************************************************************/
```

下面是函数头部的英文注释：

```
/*********************************************************************
* Function:       // 函数名称（注释对齐）
* Description:    // 函数功能、性能等的描述
* Table Accessed:     //（可选）被访问的表，此项仅对于有数据库操作的程序
* Table Updated: //（可选）被修改的表，此项仅对于有数据库操作的程序
* Input:          // 输入参数说明，包括每个参数的作用、取值说明以及参数间
                     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
* Modify Date    Version    Author          Modification
* -----------------------------------------------
* 2002/08/01     V1.0       XXXX            XXXX
*********************************************************************/
```

【规则 3-6】包含在{ }中代码块的结束处应加注释，便于阅读。特别是多分支、多重嵌套的条件语句或循环语句。

说明：此时注释可以用英文，方便查找对应的语句。

正例：

```
void Main ( )
{
   if (...)
   {
       …
       while (...)
       {
           …
       }  /* end of while (...) */   // 指明该条 while 语句结束
       …
   }  /* end of if (...) */           // 指明是哪条语句结束
}  /*  end of void main ( ) */       // 指明函数的结束
```

【规则 3-7】保证代码和注释的一致性。修改代码同时修改相应的注释，不再有用的注释要删除。

【规则 3-8】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。

说明：在使用缩写时或之前，应对缩写进行必要的说明。

正例：

如下书写比较结构清晰

```
/* 获得子系统索引 */
iSubSysIndex = aData[iIndex].iSysIndex;

/* 代码段 1 注释 */
[ 代码段 1 ]

/* 代码段 2 注释 */
[ 代码段 2 ]
```

反例 1：

如下例子注释与描述的代码相隔太远。

```
/* 获得子系统索引 */

iSubSysIndex = aData[iIndex].iSysIndex;
```

反例 2：

如下例子注释不应放在所描述的代码下面。

```
iSubSysIndex = aData[iIndex].iSysIndex;
/* 获得子系统索引 */
```

反例 3：

如下例子，显得代码与注释过于紧凑。

```
/* 代码段 1 注释 */
[ 代码段 1 ]
/* 代码段 2 注释 */
[ 代码段 2 ]
```

【规则 3-9】全局变量要有详细的注释，包括对其功能、取值范围、访问信息及访问时注意事项等的说明。

正例：

```
/*
* 变量作用：（错误状态码）
* 变量范围：例如 0 - SUCCESS    1 - Table error
* 访问说明：（访问的函数以及方法）
*/
BYTE g_ucTranErrorCode;
```

【规则 3-10】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。

正例：

```
如下注释结构比较清晰
int DoSomething(void)
{
    /* 代码段 1 注释 */
    [ 代码段 1 ]

    /* 代码段 2 注释 */
    [ 代码段 2 ]
}
```

反例：

```
如下例子，排版不整齐，阅读不方便；
int DoSomething(void)
{
/* 代码段 1 注释 */
    [ 代码段 1 ]

/* 代码段 2 注释 */
    [ 代码段 2 ]
}
```

【规则 3-11】对分支语句（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殊功能的关键，对于维护人员来说，良好的注释有助于更好的理解程序，有时甚至优于看设计文档。

〖建议 3-1〗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结构，使代码成为自注释的。

说明：清晰准确的函数、变量命名，可增加代码的可读性，减少不必要的注释。

〖建议 3-2〗尽量避免在注释中使用缩写，特别是不常用缩写。

说明：在使用缩写时，应对缩写进行必要的说明。

6 命名规则

好的命名规则能极大地增加可读性和可维护性。同时，对于一个有上百个人共同完成的大项目来说，统一命名约定也是一项必不可少的内容。本章对程序中的所有标识符（包括变量名、常量名、函数名、类名、结构名、宏定义等）的命名做出约定。

【规则 4-1】标识符要采用英文单词或其组合，便于记忆和阅读，切忌使用汉语拼音来命名。

说明：标识符应当直观且可以拼读，可望文知义，避免使人产生误解。程序中的英文单词一般不要太复杂，用词应当准确。

【规则 4-2】标识符只能由 26 个英文字母，10 个数字，及下划线的一个子集来组成，并严格禁止使用连续的下划线，下划线也不能出现在标识符头或结尾（预编译开关除外）。

说明：这样做的目的是为了使程序易读。因为 variable_name 和 variable__name 很难区分，下划线符号‘_’若出现在标识符头或结尾，容易与不带下划线‘_’的标识符混淆。

【规则 4-3】标识符的命名应当符合“min-length && max-information”原则。

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，常用单词的缩写必须统一。协议中的单词的缩写与协议保持一致。对于某个系统使用的专用缩写应该在某处做统一说明。

正例：如下单词的缩写能够被大家认可：

```
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  ;
```

规定的常用缩写如下：

常用词	缩写
Argument	Arg
Buffer	Buf
Clear	Clr
Clock	Clk
Compare	Cmp
Configuration	Cfg
Context	Ctx
Delay	Dly
Device	Dev
Disable	Dis
Display	Disp
Enable	En
Error	Err
Function	Fnct
Hexadecimal	Hex
High Priority Task	HPT
I/O System	IOS
Initialize	Init
Mailbox	Mbox
Manager	Mgr
Maximum	Max
Message	Msg
Minimum	Min
Multiplex	Mux
Operating System	OS
Overflow	Ovf
Parameter	Param
Pointer	Ptr
Previous	Prev
Priority	Prio
Read	Rd
Ready	Rdy
Register	Reg
Schedule	Sched
Semaphore	Sem

Stack	Stk
Synchronize	Sync
Timer	Tmr
Trigger	Trig
Write	Wr

【规则 4-4】程序中不要出现仅靠大小写区分的相似的标识符。

【规则 4-5】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

【规则 4-6】宏、常量名都要使用大写字母，用下划线 ‘_’ 分割单词。预编译开关的定义使用下划线 ‘_’ 开始。

正例：如 DISP_BUF_SIZE、MIN_VALUE、MAX_VALUE 等等。

【规则 4-7】变量名长度应小于 31 个字符，以保持与 ANSI C 标准一致。不得取单个字符（如 i、j、k 等）作为变量名，但是局部循环变量除外。

说明：变量，尤其是局部变量，如果用单个字符表示，很容易出错（如 l 误写成 1），而编译时又检查不出，则有可能增加排错时间。过长的变量名会增加工作量，会使程序的逻辑流程变得模糊，给修改带来困难，所以应当选择精炼、意义明确的名字，才能简化程序语句，改善对程序功能的理解。

【规则 4-8】程序中局部变量不要与全局变量重名。

说明：尽管局部变量和全局变量的作用域不同而不会发生语法错误，但容易使人误解。

【规则 4-9】使用一致的前缀来区分变量的作用域。

说明：变量活动范围前缀规范如下：

g_	：	全局变量
s_	：	模块内静态变量
空	：	局部变量不加范围前缀

【规则 4-10】使用一致的小写类型指示符作为前缀来区分变量的类型。

说明：常用变量类型前缀列表如下：

i	：	int
f	：	float
d	：	double
c	：	char
uc	：	unsigned char 或 BYTE
l	：	long
p	：	pointer
b	：	BOOL
h	：	HANDLE
w	：	unsigned short 或 WORD
dw	：	DWORD 或 unsigned long
a	：	数组，array of TYPE
str	：	字符串
t	：	结构类型

以上前缀可以进一步组合,在进行组合时，数组和指针类型的前缀指示符必须放在变量类型前缀的首位。

【规则 4-11】完整的变量名应由前缀+变量名主体组成，变量名的主体应当使用“名词”或者“形容词＋名词”，且首字母必须大写。

说明：各种前缀字符可能组合使用，在这种情况下，各前缀顺序为：变量作用域前缀、变量类型前缀。

正例：

```
float  g_fValue;          //类型为浮点数的全局变量
char  *pcOldChar;         //类型为字符指针的局部变量
```

【规则 4-12】函数名用大写字母开头的单词组合而成，且应当使用“动词”或者“动词＋名词”（动宾词组）。

说明：函数名力求清晰、明了，通过函数名就能够判断函数的主要功能。函数名中不同意义字段之间不要用下划线连接，而要把每个字段的首字母大写以示区分。函数命名采用大小写字母结合的形式，但专有名词不受限制。

【规则 4-13】结构名、联合名、枚举名由前缀 T_ 开头。

【规则 4-14】事件名由前缀 EV_ 开头。

【规则 4-15】类名采用大小写结合的方法。在构成类名的单词之间不用下划线，类名在开头加上 C，类的成员变量统一在前面加 m_ 前缀。

说明：C++Builder 中的类名在开头加 T。

正例：

```
void Object::SetValue(int iWidth, int iHeight)
{
    m_iWidth  = iWidth;
    m_iHeight = iHeight;
}
```

〖建议 4-1〗尽量避免名字中出现数字编号，如 Value1、Value2 等，除非逻辑上的确需要编号。

〖建议 4-2〗标识符前最好不加项目、产品、部门的标识。

说明：这样做的目的是为了代码的可重用性。

7 变量、常量与类型

变量、常量和数据类型是程序编写的基础，它们的正确使用直接关系到程序设计的成败，变量包括全局变量、局部变量和静态变量，常量包括数据常量和指针常量，类型包括系统的数据类型和自定义数据类型。本章主要说明变量、常量与类型使用时必须遵循的规则和一些需注意的建议，关于它们的命名，参见命名规则。

7.1 变量与常量

【规则 5-1-1】定义全局变量时必须仔细分析，明确其含义、作用、取值范围及与其它全局变量间的关系。

说明：全局变量关系到程序的结构框架，对于全局变量的理解关系到对整个程序能否正确理解，所以在对全局变量声明的同时，应对其含义、作用及取值范围进行详细地注释说明，若有必要还应说明与其它变量的关系。

【规则 5-1-2】明确全局变量与操作此全局变量的函数或过程的关系。

说明：全局变量与函数的关系包括：创建、修改及访问。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。

【规则 5-1-3】一个变量有且只有一个功能，不能把一个变量用作多种用途。

说明：一个变量只用来表示一个特定功能，不能把一个变量作多种用途，即同一变量取值不同时，其代表的意义也不同。

正例：

```
WORD    DelRelTimeQue(T_TCB *ptTcb )
{
    WORD wValue;
    WORD wLocate;

    wLocate     =   3;
    wValue      =   DeleteFromQue(wLocate);

    return wValue;
}
```

反例：

```
WORD DelRelTimeQue(T_TCB *ptTcb)
```

```
{
    WORD wLocate;

    wLocate = 3;
    wLocate =   DeleteFromQue(wLocate); // wLocate   具有两种功能。
    return wLocate;
}
```

【规则 5-1-4】循环语句与判断语句中，不允许对其它变量进行计算与赋值。

说明：循环语句只完成循环控制功能，if 语句只完成逻辑判断功能，不能完成计算赋值功能。

正例：

```
    do
{
    [处理语句]
    cInput = GetChar();
} while (cInput == 0);
```

反例：

```
do
    {
    [处理语句]
} while (cInput = GetChar());
```

【规则 5-1-5】宏定义中如果包含表达式或变量，表达式和变量必须用小括号括起来。

说明：在宏定义中，对表达式和变量使用括号，可以避免可能发生的计算错误。

正例：

```
#define  HANDLE(A, B)   (( A ) / ( B ))
```

反例：

```
#define  HANDLE(A, B)   (A / B)
```

【规则 5-1-6】使用宏定义多行语句时，必须使用 { } 把这些语句括起来。

说明：在宏定义中，对多行语句使用大括号，可以避免可能发生的错误。

〖建议 5-1-1〗尽量构造仅有一个模块或函数可以修改、创建的全局变量，而其余有关模块或函数只能访问。

说明：减少全局变量操作引起的错误。

正例：在源文件中，可按如下注释形式说明。

```
T_Student     *g_ptStudent;

变量          关系          函数

g_pStudent    创建          SystemInit(void)

              修改          无

              访问          StatScore(const T_Student *ptStudent)
                            PrintRec(const T_Student *ptStudent)
```

〖建议 5-1-2〗对于全局变量通过统一的函数访问。

说明：可以避免访问全局变量时引起的错误。

正例：

```
T_Student     g_tStudent;
    T_Student GetStudentValue(void)
{
    T_Student tStudentValue;
    [获取 g_tStudent 的访问权]
    tStudentValue = g_tStudent;
    [释放 g_tStudent 的访问权]
    return tStudentValue;
    }
BYTE SetStudentValue(const T_Student    *ptStudentValue)
{
    BYTE ucIfSuccess;
    ucIfSuccess = 0;
    [获取 g_tStudent 的访问权]
    g_tStudent = *ptStudentValue ;
    [释放 g_tStudent 的访问权]
    return ucIfSuccess;
}
```

〖建议 5-1-3〗尽量使用 const 说明常量数据，对于宏定义的常数，必须指出其类型。

正例：

```
const  int    MAX_COUNT = 1000;
#define       MAX_COUNT  (int)1000
```

反例：

```
#define       MAX_COUNT    1000
```

〖建议 5-1-4〗最好不要在语句块内声明局部变量。

7.2 类型

【规则 5-2-1】结构和联合必须被类型化。

正例：

```
typedef struct
{
    char     acName[NAME_SIZE];
    WORD  wScore;
} T_Student;
T_Student *ptStudent;
```

反例：

```
struct student
{
    char    acName[NAME_SIZE];
    WORD    wScore;
} *ptStudent;
```

〖建议 5-2-1〗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：使用统一的自定义数据类型，有利于程序的移植。

自定义数据类型	类型说明	类型定义（以 Win32 为例）
VOID	空类型	void
BOOLEAN	逻辑类型 (TRUE 或 FALSE)	unsigned char
BYTE/ UCHAR	无符号 8 位整数	unsigned char
CHAR	有符号 8 位整数	signed char

WORD16/ WORD	无符号 16 位整数	unsigned short
SWORD16/SHORT	有符号 16 位整数	signed short
WORD32/DWORD	无符号 32 位整数	unsigned int
SWORD32/INT/LONG	有符号 32 位整数	signed int
FP32/FLOAT	32 位单精度符点数	float
FP64/DOUBLE	64 位双精度符点数	double

〖建议 5-2-2〗结构是针对一种事务的抽象，功能要单一，不要设计面面俱到的数据结构。

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。

正例：

```
typedef struct TeacherStruct
{
    BYTE  aucName[8];
    BYTE  ucSex;
}T_Teacher;

typedef struct StudentStruct
{
    BYTE  ucName[8];
    BYTE  ucAge;
    BYTE  ucSex;
    WORD  wTeacherInd;
}T_Student;
```

反例：

如下结构不太清晰、合理。

```
typedef struct StudentStruct
{
    BYTE  aucName[8];
    BYTE  ucAge;
    BYTE  ucSex;
    BYTE  aucTeacherName[8];
    BYTE  ucTeacherSex;
}T_Student;
```

〖建议 5-2-3〗不同结构间的关系要尽量简单，若两个结构间关系较复杂、密切，那么应合为一个结构。

说明：两个结构关系复杂时，它们可能反映的是一个事物的不同属性。

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。

正例：

```
typedef struct PersonStruct
{
    BYTE aucName[8];
    BYTE aucAddr[40];
    BYTE ucSex;
    BYTE aucCity[15];
    BYTE ucTel;
}T_Person;
```

反例：如下两个结构的构造不合理。

```
typedef struct PersonOneStruct
{
    BYTE aucName[8];
    BYTE aucAddr[40];
    BYTE ucSex;
    BYTE ucCity[15];
}T_PersonOne;

typedef struct PersonTwoStruct
{
    BYTE aucName[8];
    BYTE aucAddr[40];
    BYTE ucTel;
}T_PersonTwo;
```

〖建议 5-2-4〗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。

正例：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 PersonBaseInfoStruct
{
    BYTE aucName[8];
```

```
    BYTE ucAge;
    BYTE ucSex;
}T_PersonBaseInfo;

typedef struct PersonAddressStruct
{
    BYTE aucAddr[40];
    BYTE aucCity[15];
    BYTE ucTel;
} T_PersonAddress;

typedef struct PersonStruct
{
    T_PersonBaseInfo  tPersonBase;
    T_PersonAddress   tPersonAddr;
} T_Person;
```

〖建议 5-2-5〗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，对于结构中未用的位明确地给予保留。

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。

正例：如下形式，不仅可节省字节空间，可读性也变好了。

```
typedef struct ExampleStruct
{
    BYTE ucValid: 1;
    BYTE ucSetFlg: 1;
    BYTE ucOther: 6;  // 保留位
    T_Person  tPerson;
}T_Example;
```

反例：如下结构中的位域排列，将占较大空间，可读性也稍差。

```
typedef struct ExampleStruct
{
    BYTE    ucValid: 1;
    T_Person  tPerson;
    BYTE    ucSetFlg: 1;
} T_Example;
```

〖建议 5-2-6〗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。

〖建议 5-2-7〗注意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。注意这些细节对程序质量的保证非常重要。

〖建议 5-2-8〗合理地设计数据并使用自定义数据类型，尽量减少没有必要的数据类型默认转换与强制转换。

〖建议 5-2-9〗当声明数据结构时，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。

正例：假如有如下短整数及结构。

```
WORD wExam;
typedef struct ExamBitStruct
{                    /* Intel 68360      */
    WORD wA1: 1;    /* bit  0      2   */
    WORD wA2: 1;    /* bit  1      1   */
    WORD wA3: 1;    /* bit  2      0   */
    WORD wOther: 13;
} T_ExamBit;
```

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）

wExam　wExam 低字节　wExam 高字节

内存：　　0 bit　　1 bit　　2 bit　...　（字节的各“位”）

T_ExamBit　　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）

```
wExam   wExam 高字节   wExam 低字节
内存：          0 bit     1 bit       2 bit     ...   （字节的各“位”）
T_ExamBit       A3          A2          A1
```

〖建议 5-2-10〗结构定义时，尽量做到 pack 1，2，4，8 无关。

说明：全局紧缩对齐可能会导致代码效率下降。

8 表达式与语句

表达式是语句的一部分，它们是不可分割的。表达式和语句虽然看起来比较简单，但使用时隐患比较多。本章归纳了正确使用表达式和 if、for、while、goto、switch 等基本语句的一些规则与建议。在写表达式和语句的时候要注意运算符的优先级，C/C++语言的运算符有数十个，运算符的优先级与结合律如下表所示。

运算符的优先级与结合律表

优先级	运算符	结合律
从高到低排列	( ) [ ] -> .	从左至右
	! ~ ++ -- （类型） sizeof + - * &	从右至左
	* / %	从左至右
	+ -	从左至右
	<< >>	从左至右
	< <= > >=	从左至右
	== !=	从左至右
	&	从左至右
	^	从左至右
	|	从左至右
	&&	从左至右
	||	从右至左
	?:	从右至左
	= += -= *= /= %= &= ^= |= <<= >>=	从左至右

【规则 6-1】一条语句只完成一个功能。

说明：复杂的语句阅读起来，难于理解，并容易隐含错误。变量定义时，一行只定义一个变量。

正例：

```
int  iHelp;
```

```
int  iBase;
int  iResult;

iHelp   = iBase;
iResult = iHelp + GetValue(&iBase);
```

反例：

```
int iBase, iResult;                    // 一行定义多个变量

iResult = iBase + GetValue(&iBase); // 一条语句实现多个功能，iBase 有两种用途。
```

【规则 6-2】在表达式中使用括号，使表达式的运算顺序更清晰。

说明：由于将运算符的优先级与结合律熟记是比较困难的，为了防止产生歧义并提高可读性，即使不加括号时运算顺序不会改变，也应当用括号确定表达式的操作顺序。

正例：

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

反例：

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

【规则 6-3】避免表达式中的附加功能，不要编写太复杂的复合表达式。

说明：带附加功能的表达式难于阅读和维护，它们常常导致错误。对于一个好的编译器，下面两种情况效果是一样的。

正例 ：

```
aiVar[1] = aiVar[2] + aiVar[3];
aiVar[4]++;
iResult = aiVar[1] + aiVar[4];
aiVar[3]++;
```

反例：

```
iResult = (aiVar[1] = aiVar[2] + aiVar[3]++) + ++aiVar[4] ;
```

【规则 6-4】不可将布尔变量和逻辑表达式直接与 TRUE、FALSE 或者 1、0 进行比较。

说明：TURE 和 FALSE 的定义值是和语言环境相关的，且可能会被重定义的。

正例：

设 bFlag 是布尔类型的变量

```
if (bFlag)      // 表示 flag 为真
if (!bFlag)     // 表示 flag 为假
```

反例：

设 bFlag 是布尔类型的变量

```
if (bFlag == TRUE)
if (bFlag == 1)
if (bFlag == FALSE)
if (bFlag == 0)
```

【规则 6-5】在条件判断语句中，当整型变量与 0 比较时，不可模仿布尔变量的风格，应当将整型变量用“==”或“!=”直接与 0 比较。

正例：

```
if (iValue == 0)

if (iValue != 0)
```

反例：

```
if (iValue)      // 会让人误解 iValue 是布尔变量

if (!iValue)
```

【规则 6-6】不可将浮点变量用“==”或“!=”与任何数字比较。

说明：无论是 float 还是 double 类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“!=”与数字比较，应该转化成“>=”或“<=”形式。

正例：

```
if ((fResult >= -EPSINON) && (fResult <= EPSINON))
```

反例：

```
if (fResult == 0.0)    // 隐含错误的比较
```

其中 EPSINON 是允许的误差（即精度）。

【规则 6-7】应当将指针变量用“==”或“!=”与 NULL 比较。

说明：指针变量的零值是“空”（记为 NULL），即使 NULL 的值与 0 相同，但是两者意义不同。

正例：

```
if (pHead == NULL)  // pHead 与 NULL 显式比较，强调 pHead 是指针变量
```

```
if (pHead != NULL)
```

反例：

```
if (pHead == 0)          // 容易让人误解 pHead 是整型变量
if (pHead != 0)
```

或者

```
if (pHead)           // 容易让人误解 pHead 是布尔变量
if (!pHead)
```

【规则 6-8】在 switch 语句中，每一个 case 分支必须使用 break 结尾，最后一个分支必须是 default 分支。

说明：避免漏掉 break 语句造成程序错误。同时保持程序简洁。

对于多个分支相同处理的情况可以共用一个 break，但是要用注释加以说明。

正例：

```
switch (iMessage)
{
     case SPAN_ON:
     {
          [处理语句]
          break;
     }
    case SPAN_OFF:
    {
          [处理语句]
          break;
    }
    default:
    {
           [处理语句]
    }
}
```

【规则 6-9】不可在 for 循环体内修改循环变量，防止 for 循环失去控制。

〖建议 6-1〗循环嵌套次数不大于 3 次。

〖建议 6-2〗do while 语句和 while 语句仅使用一个条件。

说明：保持程序简洁。如果需要判断的条件较多，建议用临时布尔变量先计算是否满足条件。

正例：

```
BOOLEAN bCondition;

do
{
    …….
    bCondition = ((tAp[iPortNo].bStateAcpActivity != PASSIVE)
              || (tAp[iPortNo].bStateLacpActivity != PASSIVE))
              && (abLacpEnabled[iPortNo])
              && (abPortEenabled[iPortNo])

} while (bCondition);
```

〖建议 6-3〗当 switch 语句的分支比较多时，采用数据驱动方式。

说明：当 switch 语句中 case 语句比较多时，会降低程序的效率。

正例：

```
extern void  TurnState(void);
extern void  SendMessage (void);
…….
void  (*StateChange[20])() = {TurnState, SendMessage, NULL, TurnState... };
…
if (StateChange[iState])
{
    (*StateChange[iState])();
}
```

〖建议 6-4〗如果循环体内存在逻辑判断，并且循环次数很大，宜将逻辑判断移到循环体的外面。

说明：下面两个示例中，反例比正例多执行了 NUM -1 次逻辑判断。并且由于前者总要进行逻辑判断，使得编译器不能对循环进行优化处理，降低了效率。如果 NUM 非常大，最好采用正例的写法，可以提高效率。

```
const  int NUM = 100000;
```

正例：

```
if (bCondition)
{
    for (i = 0; i < NUM; i++)
    {
        DoSomething();
```

```
        }
    }
    else
    {
        for (i = 0; i < NUM; i++)
        {
            DoOtherthing();
        }
    }
```

反例：

```
    for (i = 0; i < NUM; i++)
    {
        if (bCondition)
        {
            DoSomething();
        }
        else
        {
            DoOtherthing();
        }
    }
```

〖建议 6-5〗for 语句的循环控制变量的取值采用“半开半闭区间”写法。

说明：这样做更能适应 c 语言数组的特点，c 语言的下标属于一个“半开半闭区间”。

正例：

```
    int  aiScore[NUM];
    …
    for (i = 0; i < NUM; i++)
    {
        printf(“%d\n”,aiScore[i])
    }
```

反例：

```
    int  aiScore[NUM];
    …
    for (i = 0; i <= NUM-1; i++)
    {
        printf(“%d\n”,aiScore[i]);
    }
```

相比之下，正例的写法更加直观，尽管两者的功能是相同的。

〖建议 6-6〗在进行“==”比较时，将常量或常数放在“==”号的左边。

说明：可以采用这种方式，让编译器去发现错误。

正例：

```
if (NULL == pTail)
if (0 == iSum)
```

示例中有意把 p 和 NULL 颠倒。编译器认为 if (pTail = NULL) 是合法的，但是会指出 if (NULL = pTail)是错误的，因为 NULL 不能被赋值。

9 函数与过程

函数是 C/C++程序的基本功能单元。如何编写出正确、高效、易维护的函数是软件编码质量控制的关键。一个函数包括函数头，函数名，函数体，参数，返回值。其中函数头的编写参见第三章注释，函数名参见第四章命名规则，本章着重描述作为接口要素的参数和返回值，函数体的实现以及函数相互之间的调用关系。

9.1 参数

【规则 7-1-1】如果函数没有参数，则用 void 填充。

说明：函数在说明的时候，可以省略参数名。但是为了提高代码的可读性，要求不能省略。

正例：

```
void SetValue(int iWidth, int iHeight);
float GetValue(void);
```

反例：

```
void SetValue(int, int);
float GetValue();
```

【规则 7-1-2】如果参数是指针，且仅作输入用，则应在类型前加 const。

说明：防止该指针在函数体内被意外修改。

正例：

```
int GetStrLen(const char *pcString);
```

【规则 7-1-3】当结构变量作为参数时，应传送结构的指针而不传送整个结构体，并且不得修改结构中的元素，用作输出时除外。

说明：一个函数被调用的时候，形参会被一个个压入被调函数的堆栈中，在函数调用结束以后再弹出。一个结构所包含的变量往往比较多，直接以一个结构为参数，压栈出栈的内容就会太多，不但占用堆栈空间，而且影响代码执行效率，如果使用不当还可能导致堆栈的溢出。如果使用结构的指针作为参数，因为指针的长度是固定不变的，结构的大小就不会影响代码执行的效率，也不会过多地占用堆栈空间。

〖建议 7-1-1〗避免函数有太多的参数，参数个数尽量控制在 5 个以内。

说明：如果参数太多，在使用时容易将参数类型或顺序搞错，而且调用的时候也不方便。如果参数的确比较多，而且输入的参数相互之间的关系比较紧密，不妨把这些参数定义成一个结构，然后把结构的指针当成参数输入。

〖建议 7-1-2〗参数的顺序要合理。

说明：参数的顺序要遵循程序员的习惯。如输入参数放在前面，输出参数放在后面等。

正例：

```
int RelRadioChan(const T_RelRadioChanReq *ptReq, T_RelRadioChanAck *ptAck);
```

〖建议 7-1-3〗尽量不要使用类型和数目不确定的参数。

说明：对于参数个数可变的函数调用，编译器不作类型检查和参数检查。这种风格的函数在编译时丧失了严格的类型安全检查。

〖建议 7-1-4〗避免使用 BOOLEAN 参数。

说明：一方面因为 BOOLEAN 参数值无意义，TRUE/FALSE 的含义是非常模糊的，在调用时很难知道该参数到底传达的是什么意思；其次 BOOLEAN 参数值不利于扩充。

9.2 返回值

【规则 7-2-1】不要省略返回值的类型，如果函数没有返回值，那么应声明为 void 类型。

说明：C 语言中，凡不加类型说明的函数，一律自动按整型处理。如果不注明类型，容易被误解为 void 类型，产生不必要的麻烦。

C++语言有很严格的类型安全检查，不允许上述情况发生。由于 C++程序可以调用 C 函数，为了避免混乱，规定任何 C/ C++函数都必须有类型。

【规则 7-2-2】对于有返回值的函数，每一个分支都必须有返回值。

说明：为了保证对被调用函数返回值的判断，有返回值的函数中的每一个退出点都需要有返回值。

〖建议 7-2-1〗如果返回值表示函数运行是否正常，规定 0 为正常退出，不同非 0 值标识不同异常退出。避免使用 TRUE 或 FALSE 作为返回值。

正例：

```
int SubFunction(void);
```

反例：

```
BOOLEAN SubFunction(void);
```

9.3 内部实现

函数体的实现并不是随心所欲，而是有一定的规矩可循。不但要仔细检查入口参数的有效性和精心设计返回值，还要保证函数的功能单一，具有很高的功能内聚性，尽量减少函数之间的耦合，方便调试和维护。

【规则 7-3-1】对输入参数的正确性和有效性进行检查。

说明：很多程序错误是由非法参数引起的，我们应该充分理解并正确处理来防止此类错误。

【规则 7-3-2】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。

正例：

```
void SumData(int iNum, int *piData, int *piSum )
{
    int iCount ;
    int iSumTmp;          // 存储“和”的临时变量

    iSumTmp = 0;
    for (iCount = 0; iCount < iNum; iCount++)
    {
        iSumTmp += piData[iCount];
    }
    *piSum = iSumTmp;
}
```

反例：

```
void SumData(int iNum, int *piData, int *piSum )
{
    int iCount;

    *piSum = 0;
    for (iCount = 0; iCount < iNum; iCount++ )
    {
        *piSum += piData[iCount];   // piSum 成了工作变量，不好。
    }
}
```

〖建议 7-3-1〗尽量避免函数带有“记忆”功能。函数的输出应该具有可预测性，即相同的输入应当产生相同的输出。

说明：带有“记忆”功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某种“记忆状态”。这样的函数既不易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“记忆”存储器。建议尽量少用 static 局部变量，除非必需。

〖建议 7-3-2〗函数的功能要单一，不要设计多用途的函数。

说明：多用途的函数往往通过在输入参数中有一个控制参数，根据不同的控制参数产生不同的功能。这种方式增加了函数之间的控制耦合性，而且在函数调用的时候，调用相同的一个函数却产生不同的效果，降低了代码的可读性，也不利于代码调试和维护。

正例：

以下两个函数功能清晰：

```
int Add(int iParaOne, int iParaTwo)
{
    return (iParaOne + iParaTwo);
}

int Sub(int iParaOne, int iParaTwo)
{
    return (iParaOne – iParaTwo);
}
```

反例：

如果把这两个函数合并在一个函数中，通过控制参数决定结果，不可取。

```
int AddOrSub(int iParaOne, int iParaTwo, unsigned char ucAddOrSubFlg)
```

```
{
    if (INTEGER_ADD == ucAddOrSubFlg)   // 参数标记为“求和”
    {
        return (iParaOne + iParaTwo);
    }
    else
    {
        return (iParaOne –iParaTwo);
    }
}
```

〖建议 7-3-3〗函数功能明确，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。

正例：

矩形的长、宽与点的坐标基本没有任何关系，应该在不同的函数中实现。

```
void InitRect(void)
{
    // 初始化矩形的长与宽
    tRect.wLength = 0;
    tRect.wWidth  = 0;
}

void InitPoint(void)
{
    // 初始化“点”的坐标
    tPoint.wX = 10;
    tPoint.wY = 10;
}
```

反例：

矩形的长、宽与点的坐标基本没有任何关系，故以下函数是随机内聚。

```
void InitVar(void)
{
    // 初始化矩形的长与宽
    tRect.wLength = 0;
```

```
    tRect.wWidth  = 0;

    // 初始化“点”的坐标
    tPoint.wX = 10;
    tPoint.wY = 10;
}
```

〖建议 7-3-4〗函数体的规模不能太大，尽量控制在 200 行代码之内。

说明：冗长的函数不利于调试，可读性差。

〖建议 7-3-5〗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，但使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。

正例：

如下显得很清晰。

```
int Max(int iParaOne, int iParaTwo)
{
    int iMaxValue;

    iMaxValue = (iParaOne > iParaTwo) ? iParaOne : iParaTwo;
    return iMaxValue;
}
```

反例：

如下语句的功能不很明显。

```
iMaxValue = (iParaOne > iParaTwo) ? iParaOne : iParaTwo;
```

9.4 函数调用

【规则 7-4-1】必须对所调用函数的错误返回值进行处理。

说明：函数返回错误，往往是因为输入的参数不合法，或者此时系统已经出现了异常。如果不对错误返回值进行必要的处理，会导致错误的扩大，甚至导致系统的崩溃。

正例：

在程序中定义了一个函数：

```
int DbAccess(WORD wEventNo, T_InPara *ptInParam, T_OutPara *ptOutParam);
```

在引用该函数的时候应该如下处理：

```
int iResult;
```

```
iResult = DbAccess(EV_GETRADIOCHANNEL, ptReq, ptAck);
switch (iResult)
{
    case NO_CHANNEL:      // 无可用无线资源
    {
        [异常处理]
        break;
    }
    case CELL_NOTFOUND:   // 小区未找到
    {
        [异常处理]
        break;
    }
    default:
    {
        [其它处理]
    }
}
[正常处理]
```

反例：

对上面的正例中定义的函数进行如下的处理就不合适。

```
DbAccess(EV_GETRADIOCHANNEL, ptReq, ptAck);
[正常处理]
```

〖建议 7-4-1〗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。

对于前台软件为了系统的稳定性和可靠性，往往规定了进程的堆栈大小。如果采用了递归算法，收敛的条件又往往难以确定，很容易使得进程的堆栈溢出，破坏系统的正常运行；另外，由于无法确定递归的次数，降低了系统的稳定性和可靠性。

〖建议 7-4-2〗设计高扇入、合理扇出的函数。

说明：扇出是指一个函数直接调用（控制）其它函数的数目，而扇入是指有多少上级函数调用它。

扇出过大，表明函数过分复杂，需要控制和协调过多的下级函数；而扇出过小，如总是 1，表明函数的调用层次可能过多，这样不利于程序阅读和函数结构的分析，并且

程序运行时会对系统资源如堆栈空间等造成压力。函数较合理的扇出（调度函数除外）通常是 3-5。扇出太大，一般是由于缺乏中间层次，可适当增加中间层次的函数。扇出太小，可把下级函数进一步分解成多个函数，或合并到上级函数中。当然分解或合并函数时，不能改变要实现的功能，也不能违背函数间的独立性。

扇入越大，表明使用此函数的上级函数越多，这样的函数使用效率高，但不能违背函数间的独立性而单纯地追求高扇入。公共模块中的函数及底层函数应该有较高的扇入。

较好的软件结构通常是顶层函数的扇出较高，中层函数的扇出较少，而底层函数则扇入到公共模块中。

10 可靠性

为保证代码的可靠性，编程时请遵循如下基本原则，优先级递减：

- 正确性，指程序要实现设计要求的功能。
- 稳定性、安全性，指程序稳定、可靠、安全。
- 可测试性，指程序要方便测试。
- 规范/可读性，指程序书写风格、命名规则等要符合规范。
- 全局效率，指软件系统的整体效率。
- 局部效率，指某个模块/子模块/函数的本身效率。
- 个人表达方式/个人方便性，指个人编程习惯。

10.1 内存使用

【规则 8-1-1】在程序编制之前，必须了解编译系统的内存分配方式，特别是编译系统对不同类型的变量的内存分配规则，如局部变量在何处分配、静态变量在何处分配等。

【规则 8-1-2】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。

正例：

```
const int MAX_USE_NUM = 10                              // 用户号为 1-10
unsigned char aucLoginFlg[MAX_USR_NUM]={0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

void ArrayFunction(void)
{
    unsigned char ucUserNo;
    for (ucUserNo = 0; ucUserNo < MAX_USE_NUM; ucUserNo++)
```

```
    {
        aucLoginFlg[ucUser_No] = ucUserNo;
        … …
    }
}
```

反例：

```
const int MAX_USE_NUM = 10                          // 用户号为 1-10
unsigned char aucLoginFlg[MAX_USR_NUM]={0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
void ArrayFunction(void)
{
    unsigned char ucUserNo;
    for (ucUserNo = 1; ucUserNo < 11; ucUserNo++) // 10 已经越界了
    {
        aucLoginFlg[User_No] = ucUserNo;
        … …
    }
}
```

【规则 8-1-3】必须对动态申请的内存做有效性检查，并进行初始化；动态内存的释放必须和分配成对以防止内存泄漏，释放后内存指针置为 NULL。

说明：对嵌入式系统，通常内存是有限的，内存的申请可能会失败，如果不检查就对该指针进行操作，可能出现异常，而且这种异常不是每次都出现，比较难定位。

指针释放后，该指针可能还是指向原有的内存块，可能不是，变成一个野指针，一般用户不会对它再操作，但用户失误情况下对它的操作可能导致程序崩溃。

正例：

```
MemmoryFunction(void)

{
    unsigned char *pucBuffer = NULL;
    pucBuffer = GetBuffer(sizeof(DWORD));
    if (NULL != pucBuffer)          // 申请的内存指针必须进行有效性验证
    {
        // 申请的内存使用前必须进行初始化
        memset(pucBuffer, 0xFF, sizeof(DWORD));
    }
    ….
```

```
    FreeBuffer(pucBuffer);      // 申请的内存使用完毕必须释放
    pucBuffer = NULL;           // 申请的内存释放后指针置为空
    …
}
```

【规则 8-1-4】不使用 realloc( )。

说明：调用 realloc 对一个内存块进行扩展，导致原来的内容发生了存储位置的变化，realloc 函数既要调用 free，又要调用 malloc。执行时究竟调用哪个函数，取决于是要缩小还是扩大相应内存块的大小。

【规则 8-1-5】变量在使用前应初始化，防止未经初始化的变量被引用。

说明：不同的编译系统，定义的变量在初始化前其值是不确定的。有些系统会初始化为 0，而有些不是。

〖建议 8-1-1〗由于内存总量是有限的，软件系统各模块应约束自己的代码，尽量少占用系统内存。

〖建议 8-1-2〗在通信程序中，为了保证高可靠性，一般不使用内存的动态分配。

〖建议 8-1-3〗在往一个内存区连续赋值之前(memset,memcpy…)，应确保内存区的大小能够容纳所赋的数据。

〖建议 8-1-4〗尽量使用 memmove( )代替 memcpy( )。

说明：在源、目的内存区域发生重叠的情况下，如果使用 memcpy 可能导致重叠区的数据被覆盖。

10.2 指针使用

【规则 8-2-1】指针类型变量必须初始化为 NULL。

【规则 8-2-2】指针不要进行复杂的逻辑或算术操作。

说明：指针加一的偏移，通常由指针的类型确定，如果通过复杂的逻辑或算术操作，则指针的位置就很难确定。

【规则 8-2-3】如果指针类型明确不会改变，应该强制为 const 类型的指针，以加强编译器的检查。

说明：可以防止不必要的类型转换错误。

【规则 8-2-4】减少指针和数据类型的强制类型转化。

【规则 8-2-5】移位操作一定要确定类型。

说明：BYTE 的移位后还是 BYTE，如将 4 个字节拼成一个 long，则应先把字节转化成 long .

正例：

```
unsigned char ucMove;
unsigned long lMove;
unsigned long lTemp;

ucMove = 0xA3;
lTemp  = (unsigned long) ucMove;
lMove  = (lTemp << 8) | lTemp;              /* 用 4 个字节拼成一个长字 */
lMove  = (lMove << 16) | lMove;
```

反例：

```
unsigned char ucMove = 0xA3;
unsigned long lMove;
lMove = (ucMove << 8) | ucMove;         /* 用 4 个字节拼成一个长字 */
lMove = (lMove << 16) | lMove;
```

【规则 8-2-6】对变量进行赋值时，必须对其值进行合法性检查，防止越界等现象发生。

说明：尤其对全局变量赋值时，应进行合法性检查，以提高代码的可靠性、稳定性。

10.3 类和函数

【规则 8-3-1】类中的属性应声明为 private，用公有的函数访问。

说明：这样可以防止对类属性的误操作。

正例：

```
class CCount
{
public:
    CCount (void);
    ~ CCount (void);
    int GetCount(void);
    void SetCount(int iCount);
private:
    int m_iCount;
}
```

【规则 8-3-2】在编写派生类的赋值函数时，注意不要忘记对基类的成员变量重新赋值。

说明：除非在派生类中调用基类的赋值函数，否则基类变量不会自动被赋值。

正例：

```
class CBase
{
public:
    …
    CBase & operate = (const CBase &other);    // 类 CBase 的赋值函数
private:
    int  m_iLength;
    int  m_iWidth;
    int  m_iHeigth;
};

class CDerived : public CBase
{
public:
    …
    CDerived & operate = (const CDerived &other); // 类 CDerived 的赋值函数
private:
    int  m_iLength;
    int  m_iWidth;
```

```
    int  m_iHeigth;
};

CDerived & CDerived::operate = (const CDerived &other)
{
    if (this == &other)            //（1）检查自赋值
    {
         return *this;
    }

    CBase::operate =(other);  //（2）对基类的数据成员重新赋值
                          // 因为不能直接操作私有数据成员
                          //（3）对派生类的数据成员赋值
    m_iLength = other.m_iLength;
    m_iWidth  = other.m_iWidth;
    m_iHeigth = other.m_iHeigth;

    return *this;             //（4）返回本对象的引用
}
```

【规则 8-3-3】构造函数应完成简单有效的功能，不应完成复杂的运算和大量的内存管理。

说明：如果该类有相当多的初始化工作，应生成专门的 Init（...）函数，不能完全在构造函数中进行，因为构造函数没有返回值，不能确定初始化是否成功。

【规则 8-3-4】不要在栈中分配类的实例，也不要生成全局类实例。

说明：这里所说的类，是带有构造函数的类。在栈中分配类的实例，类的构造函数和析构函数会带来很多麻烦。而全局类实例使得用户不能对该实例进行管理。

正例：

```
void  MemmoryFunction(...)
{
    CMyClass *pMyClass = NULL;
    pMyClass  = new CMyClass(void);    // 动态申请内存

    if (pMyClass == NULL)          // 对申请的指针作有效性检查
    {
        ...
        delete pMyClass ;              // 内存使用完后应释放
```

```
        pMyClass = NULL;
        …
    }
    …
}
```

反例：

```
void MemmoryFunction(...)
{
    CMyClass  OneClass;      // 在栈分配类的实例可能导致构造函数的失败
    OneClass.Param1 = 2;     // 如果分配不成功，则对实例成员的访问是违规的
    …
}               // 在函数返回前，要调用类的析构函数，则又造成析构异常
```

【规则 8-3-5】正确处理拷贝构造函数与赋值函数。

说明：由于并非所有的对象都会使用拷贝构造函数和赋值函数，程序员可能对这两个函数有些轻视。如果不主动编写拷贝构造函数和赋值函数，编译器将以“位拷贝”的方式自动生成缺省的函数。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。

反例：

```
class CString
{
public:
    CString(const char *pStr = NULL);          // 普通构造函数
    CString(const CString &other);             // 拷贝构造函数
    ~ CString(void);                           // 析构函数
    CString & operate =(const CString &other); // 赋值函数
public:
    char  *m_pData;                                 // 用于保存字符串
};
CString::CString(const char *pStr)
{
    if (pStr == NULL)
    {
        m_pData    = new char[10];
       *m_pData    = ‘\0’;
    }
    else
    {
        int iLength;
        iLength    = strlen(pStr);
```

```
        m_pData = new char[iLength + 1];
        strcpy(m_pData, pStr);
    }
}

CString::~CString(void)    // CString 的析构函数
{
    delete [] pData;   // 由于 pData 是内部数据类型,也可以写成 delete pData;

}
main()
{
    CString  CStringA(“hello);
    CString  CStringB(“word”);
    CString  CstringC = CStringA;      // 拷贝构造函数
    CStringC = CStringB;               // 赋值函数
    CStringB.pData = CStringA. pData;  // 这将造成三个错误：
    /* 1. CStringB.m_pData 原有的内存没被释放，造成内存泄露；
    * 2. CStringB.m_pData 和 CStringA.m_pData 指向同一块内存，CStringA 或
    *CstringB
    * 任何一方变动都会影响另一方
    * 3. 对象被析构时，m_pData 被释放了两次。应把 m_pData 改成私有数据,用赋
    * 值函数进行赋值
    */
    ….
}
```

【规则 8-3-6】过程/函数中申请的（为打开文件而使用的）文件句柄，在过程/函数退出之前要关闭，除非要把这个句柄传递给其它函数使用。

〖建议 8-3-1〗编写可重入函数时，若使用全局变量，则应通过信号量（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。

正例：

假设 g_iExam 是 int 型全局变量，函数 SqureExam 返回 g_iExam 平方值。那么如下函数具有可重入性。

```
unsigned int Example(int iPara)
{
```

```
    unsigned int iTemp;

    [申请信号量操作]
    g_iExam  = iPara;
    iTemp    = SquareExam( );
    [释放信号量操作]

    return iTemp;
}
```

反例：

如下函数不具有可重入性。

```
unsigned int Example( int iPara )
{
    unsigned int iTemp;

    g_iExam  = iPara;          // 在访问全局变量前没有使用信号量保护
    iTemp    = SquareExam();

    return iTemp;
}
```

此函数若被多个进程调用的话，其结果可能是未知的，因为当访问全局变量语句刚执行完后，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 g_iExam 赋与另一个不同的 iPara 值，所以当控制重新回到“iTemp = SquareExam()”后，计算出的 iTemp 很可能不是预想中的结果。

11 可测试性

在设计阶段就必须考虑所编写代码的可测试性，只有提供足够的测试手段才能全面、高效地发现和解决代码中的各类问题。编写的代码是否可测试，是衡量代码质量的最基本的、最重要的尺度之一。

程序设计过程中（或程序编码完毕后），必须编写软件模块测试文档，测试文档的编写规范参见后续规范，主要应包括：设计思路、程序输入、程序输出和数据结构等。测试是设计的一部分。

【规则 9-1】在同一项目组或产品组内，为准备集成测试和系统联调，要有一套统一的调测开关及相应信息输出函数，并且要有详细的说明。统一的调试接口和输出函数由模块设计和测试人员根据项目特性统一制订，由项目系统人员统一纳入系统设计中

说明：本规则是针对项目组或产品组的。

【规则 9-2】在同一个项目组或产品组内，调测打印出的信息串要有统一的格式。信息串中应当包含所在的模块名（或源文件名）及行号等信息。

说明：统一的调测信息格式便于集成测试。

【规则 9-3】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）。

说明：程序的调试与测试是软件生存周期中非常重要的一个阶段，如何对软件进行较全面、高效率的测试并尽可能地找出软件中的错误就成为非常关键的问题。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，还应设计出一系列测试代码作为手段，为单元测试、集成测试及系统联调提供方便。

〖建议 9-1〗在同一项目组或产品组内，可以统一由模块设计和测试人员开发调试信息接收平台，统一对软件调试信息进行分析。

说明：本建议是针对项目组或产品组的。

〖建议 9-2〗 设计人员在编程的同时要完成调试信息输出接口函数，但是测试点的选择可以由模块测试人员根据需要合理选择，测试点的选择可以根据测试用例而定，不同的测试用例选择不同的测试点。

说明：为模块测试做准备。

〖建议 9-3〗调测开关应分为不同级别和类型。

说明：调测开关的设置及分类应从以下几方面考虑：针对模块或系统某部分代码的调测；针对模块或系统某功能的调测；出于某种其它目的，如对性能、容量等的测试。这样做便于软件功能的调测，并且便于模块的单元测试、系统联调等。

〖建议 9-4〗在进行集成测试和系统联调之前，要构造好测试环境、测试项目及测试用例，同时仔细分析并优化测试用例，以提高测试效率。

说明：好的测试用例应尽可能模拟出程序所遇到的边界值、各种复杂环境及一些极端情况等。

〖建议 9-5〗程序的编译开关应该设置为最高优先级，并且编译选项不要选择优化。

说明：将编译开关置为最高优先级，可以将程序的错误尽量暴露在编译阶段，便于修正程序；将编译选项设置为不优化，是为了避免编译器优化时出错，导致程序运行出错，也更容易在程序出错时对错误进行定位。

〖建议 9-6〗在设计时考虑以下常见发现错误的方法。

说明： 以下发现错误的方法为可以为编写可测试性代码提供思路:

- 使用所有数据建立假设
- 求精发现错误的测试用例
- 通过不同的方法再生错误
- 产生更多的数据以生成更多的假设
- 使用否定测试结果
- 提出尽可能多的假设
- 缩小可疑代码区
- 检查最近作过修改的代码
- 扩展可疑代码区
- 逐步集成
- 怀疑以前出过错的子程序
- 耐心检查
- 为迅速的草率的调试设定最大时间
- 检查一般错误
- 使用交谈调试法
- 中断对问题的思考

〖建议 9-7〗在设计时考虑以下常见改正错误的方法。

说明：以下改正错误的方法可以为编写可测试性代码提供思路：

- 理解问题的实质
- 理解整个程序
- 确诊错误
- 放松情绪
- 保存初始源代码
- 修改错误而不是修改症状
- 仅为某种原因修改代码
- 一次作一个修改
- 检查你的工作，验证修改

- 寻找相似错误

〖建议 9-8〗程序开发人员对自己模块内的函数必须通过有效的方法进行测试，保证所有代码都执行到。

12 断言与错误处理

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。在实际应用时，可根据具体情况灵活地设计断言。

【规则 10-1】整个软件系统应该采用统一的断言。如果系统不提供断言，则应该自己构造一个统一的断言供编程时使用。

说明：整个软件系统提供一个统一的断言函数，如 Assert(exp)，　同时可提供不同的宏进行定义（可根据具体情况灵活设计），如：

（1）#define ASSERT_EXIT_M中断当前程序执行，打印中断发生的文件、行号，该宏一般在单调时使用。

（2）#define ASSERT_CONTINUE_M打印程序发生错误或异常的文件,行号,继续进行后续的操作，该宏一般在联调时使用。

（3）#define ASSERT_OK_M　空操作，程序发生错误情况时，继续进行，可以通过适当的方式通知后台的监控或统计程序，该宏一般在 RELEASE 版本中使用。

【规则 10-2】使用断言捕捉不应该发生的非法情况。不要混淆非法情况与错误情况之间的区别，后者是必然存在的并且是一定要作出处理的。

说明：断言是用来处理不应该发生的错误情况的，对于可能会发生的且必须处理的情况要写防错程序，而不是断言。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，不能用断言来实现。

【规则 10-3】指向指针的指针及更多级的指针必须逐级检查。

说明：对指针逐级检查，有利于给错误准确定位。

正例：

```
Assert ( (ptStru != NULL)
       && (ptStru->ptForward != NULL)
       && (ptStru->ptForward->ptBackward != NULL));
```

反例：

```
Assert (ptStru->ptForward->ptBackward != NULL);
```

【规则 10-4】对较复杂的断言加上明确的注释。

说明：为复杂的断言加注释，可澄清断言含义并减少不必要的误用。

【规则 10-5】用断言保证没有定义的特性或功能不被使用。

说明：假设某通信模块在设计时，在消息处理接口准备处理“同步消息”和“异步消息”。但当前的版本中的消息处理接口仅实现了处理“异步消息”，且在此版本的正式发行版中，用户层（上层模块）不应产生发送“同步消息”的请求，那么在测试时可用断言检查用户是否发送了“同步消息”。

正例：

```
const CHAR ASYN_EVENT = 0;
const CHAR SYN_EVENT  = 1;

WORD MsgProcess( T_ExamMessage *ptMsg )
{
    CHAR cType;                     // 消息类型

    Assert (ptMsg != NULL);         // 用断言检查消息是否为空
    cType = GetMsgType (ptMsg);
    Assert (cType != SYN_EVENT);    // 用断言检查是否是同步消息

    ...                             // 其它代码
}
```

【规则 10-6】用调测开关来切换软件的 DEBUG 版和 RELEASE 版，而不要同时存在 RELEASE 版本和 DEBUG 版本的不同源文件，以减少维护的难度。

说明：DEBUG 版和 RELEASE 版的源文件相同，通过调测开关来进行区分，有利于版本的管理和维护。

【规则 10-7】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）。

说明：加快软件运行速度。

【规则 10-8】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。

说明：即有测试代码的软件和关掉测试代码的软件，在功能行为上应该一致。

【规则 10-9】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。

说明：对 RELEASE 版本不用的测试代码可以通过断言来检查测试代码中的非法情况。

〖建议 10-1〗用断言对程序开发环境（OS/Compiler/Hardware）的假设进行检查。

说明：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。如用断言检查编译器的 int 型数据占用的内存空间是否为 2 个字节：

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

〖建议 10-2〗尽可能模拟出各种程序出错状态，测试软件对出错状态的处理。

说明：“不要让事情很少发生。”需要确定子系统中可能发生哪些事情，并且使它们一定发生和经常发生。如果发现子系统中有极罕见的行为，要干方百计地设法使其重现。

〖建议 10-3〗编写错误处理程序，然后在处理错误之后可用断言宣布发生错误。

说明：假如某模块收到通信链路上的消息，则应对消息的合法性进行检查，若消息类别不是通信协议中规定的，则应进行出错处理，之后可用断言报告。

正例：

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

/* 注意:这个函数不终止和退出程序 t */
VOID AssertReport(CHAR *pcFileName, WORD wLineno)
{
    printf("\n[EXAM]Error Report:%s, ling%u\n",
```

```
            pcFileName, wLineno);
}

#define ASSERT_REPORT(condition)
if (condition)                    // 若条件成立，则无动作
{
    NULL;
}
else                              // 否则报告
{
    AssertReport(_FILE_, _LINE_)
}
#else                             // 若不使用断言测试

#define ASSERT_REPORT(condition)   NULL

#endif                            // 断言结束

WORD MsgHandle(CHAR cMsgname, CHAR *pcMsg)
{
    switch(cMsgname)
    {
        case MSG_ONE:
        {
            …                     // 消息 MSG_ONE 处理
            retum MSG_HANDLE_SUCCESS;
        }
        …                         // 其它合法消息处理
        default:
        {
            …                     // 消息出错处理
            ASSERT_REPORT（FALSE）; // “合法”消息不成立，报告
            retum MSG_HANDLE_ERROR;
        }
    }
}
```

〖建议 10-4〗使用断言检查函数输入参数的有效性、合法性。

说明：检查函数的输入参数是否合法，如输入参数为指针，则可用断言检查该指针是否为空，

如输入参数为索引，则检查索引是否在值域范围内。

正例：

```
BYTE  StoreCsrMsg(WORD wIndex,  T_CMServReq  *ptMsgCSR)
{
    WORD      wStoreIndex;
    T_FuncRet     tFuncRet;

    Assert (wIndex < MAX_DATA_AREA_NUM_A);  // 使用断言检查索引
    Assert (ptMsgCSR != NULL);                    // 使用断言检查指针

    …                                              // 其它代码

    return OK_M;
}
```

〖建议 10-5〗对所有具有返回值的接口函数的返回结果进行断言检查。

说明：对接口函数的返回结果进行检查，可以避免程序运行过程中使用不正确的返回值引起错误。

正例：

```
BYTE HandleTpWaitAssEvent(T_CcuData *ptUdata, BYTE *pucMsg)
{
    T_CacAssignFail   *ptAssignfail;
    T_CccData       *ptCdata;

    ptAssignfail = (T_CacAssignFail *)pbMsg;

    …                          // 其它代码

    ptCdata = GetCallData(ptUdata->waCallindex[0]);
    Assert (ptCdata != NULL);    // 使用断言对函数的返回结果进行检查

     …                         // 其它代码

     return CCNO_M;
}
```

附录 A
（资料性附录）
编程模版

A.1 头文件书写模板：

```
/*********************************************************************************
* 版权所有 (C)2001, 深圳市中兴通讯股份有限公司。
*
* 文件名称： // 文件名
* 文件标识： // 见配置管理计划书
* 内容摘要： // 简要描述本文件的内容，包括主要模块、函数及其功能的说明
* 其它说明： // 其它内容的说明
* 当前版本： // 输入当前版本
* 作　　者： // 输入作者名字
* 完成日期： // 输入完成日期，例：2000 年 2 月 25 日
*
* 修改记录 1：// 修改历史记录，包括修改日期、修改者及修改内容
*　　修改日期：
*　　版 本 号：
*　　修 改 人：
*　　修改内容：
* 修改记录 2：…
*********************************************************************************/

#ifndef COMMAND_H
#define COMMAND_H

#pragma once

#include <dos.h>
#include "mutex.h"

/*********************************************************************************
 *                              常量                                             *
 *********************************************************************************/

#define CONFIG_CODE_MIN      0x01    /* 最小命令码 */
#define CONFIG_CODE_MAX      0x4F    /* 最大命令码 */
#define SMCC_SETNECFG_NCP    0x01    /* SMCC 设置 NCP 网元属性命令 */
```

```
/*****************************************************************************
 *                              宏定义                                       *
 *****************************************************************************/

#define NcpCmd1DesAddr(bf_ptr) (*(ULONG * const)(bf_ptr)) /* 取报文目的地址 */

/*****************************************************************************
 *                              数据类型                                     *
 *****************************************************************************/

enum TimerState{Idle, Active, Done};        //计时器状态
enum TimerType{OneShot, Periodic};          //计时器类型

/*****************************************************************************
 *                              类声明                                       *
 *****************************************************************************/

class Timer         /* 定时器 */
{
public:

    TimerState    State;
    TimerType     Type;
    unsigned int  iLength;
    unsigned int  iCount;

    Timer();
    ~Timer();

    int  Start(unsigned int iMilliseconds);
    int  Waitfor(void);
    void Cancel(void);

private:

    static void Interrupt(void);
};
```

```
/*****************************************************************************
 *                                模板                                       *
 *****************************************************************************/


/*****************************************************************************
 *                              全局变量声明                                   *
 *****************************************************************************/

extern Timer g_Timeer;      /* 全局计时器 */

/*****************************************************************************
 *                              全局函数原型                                   *
 *****************************************************************************/

extern void SetBoardReset(void);     /*  设置本单板复位  */

#endif  /* COMMAND_H */
```

A.2 实现文件书写模板：

```
/*****************************************************************************
* 版权所有 (C)2001, 深圳市中兴通讯股份有限公司。
*
* 文件名称： // 文件名
* 文件标识： // 见配置管理计划书
* 内容摘要： // 简要描述本文件的内容，包括主要模块、函数及其功能的说明
* 其它说明： // 其它内容的说明
* 当前版本： // 输入当前版本
* 作     者： // 输入作者名字
* 完成日期： // 输入完成日期，例：2000 年 2 月 25 日
*
* 修改记录 1：// 修改历史记录，包括修改日期、修改者及修改内容
*     修改日期：
*     版 本 号：
*     修 改 人：
*     修改内容：
* 修改记录 2：…
*****************************************************************************/
```

```
#include        <board.h>
#include        <mpc8xx.h>
#include        "ncp.h"
#include        "timer.h"

/*****************************************************************************
 *                                常量                                       *
 *****************************************************************************/



/*****************************************************************************
 *                                宏                                         *
 *****************************************************************************/


/*****************************************************************************
 *                              数据类型                                     *
 *****************************************************************************/

typedef unsigned char TaskId;

/*****************************************************************************
 *                              全局变量                                     *
 *****************************************************************************/

TaskId Task::nextId = 0;


/*****************************************************************************
 *                              局部函数原型                                 *
 *****************************************************************************/



/*****************************************************************************
 *                    类 Timer 实现--公有部分                                *
 *****************************************************************************/
```

```
/*****************************************************************************
* 函数名称： Timer()
* 功能描述： 构造函数
* 访问的表：
* 修改的表：
* 输入参数：
* 输出参数：
* 返 回 值：
* 其它说明：
* 修改日期     版本号      修改人       修改内容
* ------------------------------------------
* 02/08/01       V1.0       XXXX       XXXX
*****************************************************************************/

Timer::Timer(void)
{

    // 初始化定时器

    // 其它初始化动作

}   /* Timer() */


/*****************************************************************************
* 函数名称： ~Timer()
* 功能描述： 析构函数
* 访问的表：
* 修改的表：
* 输入参数：
* 输出参数：
* 返 回 值：
* 其它说明：
* 修改日期     版本号      修改人       修改内容
* ------------------------------------------
* 2002/08/01  V1.0       XXXX       XXXX
*****************************************************************************/

Timer::~Timer(void)
{
```

```
    // 取消定时器

}   /* ~Timer() */


/*****************************************************************************
* 函数名称： Start(unsigned int iMilliseconds)
* 功能描述： 启动定时器
* 访问的表：
* 修改的表：
* 输入参数： unsigned int iMilliseconds
* 输出参数：
* 返 回 值： 0 成功     -1 如果定时器已经在使用
* 其它说明：
* 修改日期    版本号     修改人       修改内容
* ------------------------------------------
* 2002/08/01  V1.0      XXXX      XXXX
*****************************************************************************/

int Timer::Start(unsigned int iMilliseconds)
{

    // 启动定时器动作（略）

}   /* Start() */


/*****************************************************************************
 *           类 Timer 实现--保护部分                                         *
 *****************************************************************************/


/*****************************************************************************
 *           类 Timer 实现--私有部分                                         *
 *****************************************************************************/

/*****************************************************************************
* 函数名称： Interrupt(void)
* 功能描述： 中断处理
```

```
* 访问的表：
* 修改的表：
* 输入参数：
* 输出参数：
* 返 回 值：
* 其它说明：
* 修改日期     版本号      修改人        修改内容
* ------------------------------------------
* 2002/08/01    V1.0      XXXX       XXXX
*******************************************************************************/


void Timer::Interrupt(void)
{

    // 实现略
}   /* Interrupt() */



/*******************************************************************************
 *                          全局函数实现                                       *
 *******************************************************************************/

/*******************************************************************************
* 函数名称： void SetBoardReset(void)
* 功能描述： 设置单板复位
* 访问的表：
* 修改的表：
* 输入参数：
* 输出参数：
* 返 回 值：
* 其它说明：
* 修改日期     版本号      修改人        修改内容
* ------------------------------------------
* 2002/08/01      V1.0        XXXX       XXXX
*******************************************************************************/

void SetBoardReset(void)
{
```

```
    /* 实现略 */
}    /* SetBoardReset(void) */

/*****************************************************************************
 *                          局部函数实现                                     *
 *****************************************************************************/
```

附录 B
（资料性附录）
规范检查表

文件结构		
重要性	审查项	结论
	头文件和定义文件的名称是否合理?	
	头文件和定义文件的目录结构是否合理?	
	版权和版本声明是否完整?	
重要	头文件是否使用了 ifndef/define/endif 预处理块?	
	头文件中是否只存放“声明”而不存放“定义”	
	……	
程序的版式		
重要性	审查项	结论
	空行是否得体?	
	代码行内的空格是否得体?	
	长行拆分是否得体?	
	“{”和“}”是否各占一行并且对齐于同一列?	
重要	一行代码是否只做一件事? 如只定义一个变量,只写一条语句。	
重要	If、for、while、do 等语句自占一行，不论执行语句多少都要加“{}”。	
重要	在定义变量（或参数）时，是否将修饰符 * 和 & 紧靠变量名?	
	注释是否清晰并且必要?	
重要	注释是否有错误或者可能导致误解?	
重要	类结构的 public, protected, private 顺序是否在所有的程序中保持一致?	
	……	
命名规则		
重要性	审查项	结论
重要	命名规则是否与所采用的操作系统或开发工具的风格保持一致?	
	标识符是否直观且可以拼读?	
	标识符的长度应当符合“min-length && max-information”原则?	
重要	程序中是否出现相同的局部变量和全部变量?	

	类名、函数名、变量和参数、常量的书写格式是否遵循一定的规则？	
	静态变量、全局变量、类的成员变量是否加前缀？	
	……	
表达式与基本语句		
重要性	审查项	结论
重要	如果代码行中的运算符比较多，是否已经用括号清楚地确定表达式的操作顺序？	
	是否编写太复杂或者多用途的复合表达式？	
重要	是否将复合表达式与“真正的数学表达式”混淆？	
重要	是否用隐含错误的方式写 if 语句？例如 （1）将布尔变量直接与 TRUE、FALSE 或者 1、0 进行比较。 （2）将浮点变量用“==”或“！=”与任何数字比较。	
	如果循环体内存在逻辑判断，并且循环次数很大，是否已经将逻辑判断移到循环体的外面？	
重要	Case 语句的结尾是否忘了加 break？	
重要	是否忘记写 switch 的 default 分支？	
重要	使用 goto 语句时是否留下隐患？例如跳过了某些对象的构造、变量的初始化、重要的计算等。	
	……	
常量		
重要性	审查项	结论
	是否使用含义直观的常量来表示那些将在程序中多次出现的数字或字符串？	
	在 C++ 程序中，是否用 const 常量取代宏常量？	
重要	如果某一常量与其它常量密切相关，是否在定义中包含了这种关系？	
	是否误解了类中的 const 数据成员？因为 const 数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的。	
	……	
函数设计		
重要性	审查项	结论
	参数的书写是否完整？不要贪图省事只写参数的类型而省略参数名字。	
	参数命名、顺序是否合理？	
	参数的个数是否太多？	
	是否使用类型和数目不确定的参数？	

	是否省略了函数返回值的类型？	
	函数名字与返回值类型在语义上是否冲突？	
重要	是否将正常值和错误标志混在一起返回？正常值应当用输出参数获得，而错误标志用 return 语句返回。	
重要	在函数体的“入口处”，是否用 assert 对参数的有效性进行检查？	
重要	使用滥用了 assert？ 例如混淆非法情况与错误情况，后者是必然存在的并且是一定要作出处理的。	
重要	return 语句是否返回指向“栈内存”的“指针”或者“引用”？	
	是否使用 const 提高函数的健壮性？const 可以强制保护函数的参数、返回值，甚至函数的定义体。“Use const whenever you need”	
	……	
内存管理		
重要性	审查项	结论
重要	用 malloc 或 new 申请内存之后，是否立即检查指针值是否为 NULL？（防止使用指针值为 NULL 的内存）	
重要	是否忘记为数组和动态内存赋初值？（防止将未被初始化的内存作为右值使用）	
重要	数组或指针的下标是否越界？	
重要	动态内存的申请与释放是否配对？（防止内存泄漏）	
重要	是否有效地处理了“内存耗尽”问题？	
重要	是否修改“指向常量的指针”的内容？	
重要	是否出现野指针？例如 （1）指针变量没有被初始化。 （2）用 free 或 delete 释放了内存之后，忘记将指针设置为 NULL。	
重要	是否将 malloc/free 和 new/delete 混淆使用？	
重要	malloc 语句是否正确无误？例如字节数是否正确？类型转换是否正确？	
重要	在创建与释放动态对象数组时，new/delete 的语句是否正确无误？	
	……	
C++ 函数的高级特性		
重要性	审查项	结论
	重载函数是否有二义性？	
重要	是否混淆了成员函数的重载、覆盖与隐藏？	
	运算符的重载是否符合制定的编程规范？	
	是否滥用内联函数？例如函数体内的代码比较长，函数体	

	内出现循环。	
重要	是否用内联函数取代了宏代码？	
	……	
类的构造函数、析构函数和赋值函数		
重要性	审查项	结论
重要	是否违背编程规范而让 C++ 编译器自动为类产生四个缺省的函数：(1)缺省的无参数构造函数；(2)缺省的拷贝构造函数；(3) 缺省的析构函数；(4) 缺省的赋值函数。	
重要	构造函数中是否遗漏了某些初始化工作？	
重要	是否正确地使用构造函数的初始化表？	
重要	析构函数中是否遗漏了某些清除工作？	
	是否错写、错用了拷贝构造函数和赋值函数？	
重要	赋值函数一般分四个步骤：(1)检查自赋值；(2)释放原有内存资源；(3) 分配新的内存资源，并复制内容；(4) 返回 *this。是否遗漏了重要步骤？	
重要	是否正确地编写了派生类的构造函数、析构函数、赋值函数？注意事项： (1) 派生类不可能继承基类的构造函数、析构函数、赋值函数。 (2) 派生类的构造函数应在其初始化表里调用基类的构造函数。 (3) 基类与派生类的析构函数应该为虚（即加 virtual 关键字)。 (4) 在编写派生类的赋值函数时，注意不要忘记对基类的数据成员重新赋值。	
	……	
类的高级特性		
重要性	审查项	结论
重要	是否违背了继承和组合的规则？ (1) 若在逻辑上 B 是 A 的“一种”，并且 A 的所有功能和属性对 B 而言都有意义，则允许 B 继承 A 的功能和属性。 (2) 若在逻辑上 A 是 B 的“一部分”(a part of)，则不允许 B 从 A 派生，而是要用 A 和其它东西组合出 B。	
	……	
其它常见问题		
重要性	审查项	结论
重要	数据类型问题： (1) 变量的数据类型有错误吗？ (2) 存在不同数据类型的赋值吗？ (3) 存在不同数据类型的比较吗？	

重要	变量值问题： （1）变量的初始化或缺省值有错误吗？ （2）变量发生上溢或下溢吗？ （3）变量的精度够吗？	
重要	逻辑判断问题： （1）由于精度原因导致比较无效吗？ （2）表达式中的优先级有误吗？ （3）逻辑判断结果颠倒吗？	
重要	循环问题： （1）循环终止条件不正确吗？ （2）无法正常终止（死循环）吗？ （3）错误地修改循环变量吗？ （4）存在误差累积吗？	
重要	错误处理问题： （1）忘记进行错误处理吗？ （2）错误处理程序块一直没有机会被运行？ （3)错误处理程序块本身就有毛病吗？如报告的错误与实际错误不一致，处理方式不正确等等。 （4）错误处理程序块是“马后炮”吗？如在被它被调用之前软件已经出错。	
重要	文件 I/O 问题： （1）对不存在的或者错误的文件进行操作吗？ （2）文件以不正确的方式打开吗？ （3）文件结束判断不正确吗？ （4）没有正确地关闭文件吗？	

参考文献

[1] Steve Maguire. Writing Clean Code. Microsoft Corporation, 1998

[2] Jean J.Labrosse. C Coding Standard. Micriμm, Inc. , 1999

[3] Steve Mcconnell. Code Complete. 电子工业出版社, 1993

[4] Herbert Shildt. C 语言大全. 电子工业出版社, 1999

[5] Meyers, Scott. Effective C++中文版.华中理工大学出版社，2001

[6] 林锐. 高质量 C++编程指南.上海贝尔, 2001

[7] Brian W.Kernighan. 程序设计实践. 高等教育出版社, 2001

[8] 周之英 编著. 现代软件工程. 科学出版社，1999

[9] 上海一所. 软件编程规范. 深圳市中兴通讯股份有限公司, 2002

[10] 本部事业部. 软件编程规范. 深圳市中兴通讯股份有限公司, 2002

[11] 网络事业部. 软件编程规范. 深圳市中兴通讯股份有限公司, 2000

[12] CDMA 事业部. 统一平台软件编程规范. 深圳市中兴通讯股份有限公司, 2000

[13] 移动事业部. 前台软件编码细则. 深圳市中兴通讯股份有限公司, 2000

[14] 网络事业部. 3G 统一平台软件编程规范. 深圳市中兴通讯股份有限公司, 2002

[15] 网络事业部. Softswitch 产品部编程规范. 深圳市中兴通讯股份有限公司, 2002

[16] 技术中心. 操作系统平台 C 编程规范. 深圳市中兴通讯股份有限公司, 2002