C编程规范

说明

感谢为编程规范作出辛勤劳动的作者！

本规范总则的内容包括：排版、注释、标识符命名、变量使用、代码可测性、程序效率、质量保证、代码编译、单元测试、程序版本与维护等。

本规范总则的示例都以C语言为背景，采用以下的术语描述：

★ **规则**：编程时强制必须遵守的原则。

★ **建议**：编程时必须加以考虑的原则。

★ **说明**：对此规则或建议进行必要的解释。

★ **示例**：对此规则或建议从正、反两个方面给出例子。

目　录

1 排版

[1]1-1：程序块要采用缩进风格编写，缩进的空格数为4个。

说明：对于由开发工具自动生成的代码可以有不一致。

[1]1-2：相对独立的程序块之间、变量说明之后必须加空行。

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;
```

[1]1-3：较长的语句（>80字符）要分成多行书写，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。

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));
```

[1]1-4：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：

```
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
}
```

[1]1-5：若函数或过程中的参数较长，则要进行适当的划分。

示例：

```
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 );
```

[1]1-6：不允许把多个短语句写在一行中，即一行只写一条语句。

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;
```

[1]1-7：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;
}
```

[1]1-8：对齐只使用空格键，不使用TAB键。

说明：以免用不同的编辑器阅读程序时，因TAB键所设置的空格数目不同而造成程序布局不整齐，不要使用BC作为编辑器合版本，因为BC会自动将8个空格变为一个TAB键，因此使用BC合入的版本大多会将缩进变乱。

[1]1-9：函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格，case语句下的情况处理语句也要遵从语句缩进要求。

[1]1-10：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
}
```

[1]1-11：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（如一>），后不应加空格。

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。

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，如果语句已足够清晰则括号内侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在C/C++语言中括号已经是最清晰的标志了。

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。给操作符留空格时不要连续留两个以上空格。

示例：

(1) 逗号、分号只在后面加空格。

```
int 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 (current_time >= MAX_TIME_VALUE)
a = b + c;
a *= 2;
a = b ^ 2;
```

(3)"!"、"~"、"++"、"--"、"&"（地址运算符）等单目操作符前后不加空格。

```
*p = 'a';       // 内容操作"*"与内容之间
flag = !isEmpty; // 非操作"!"与内容之间
p = &mem;       // 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
```

(5) if、for、while、switch等与后面的括号间应加空格，使if等关键字更为突出、明显。

```
if (a >= b && c > d)
```

½1-1：一行程序以小于80字符为宜，不要写得过长。

2 注释

[1]2-1：一般情况下，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，在该加的地方都加了，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。

[1]2-2：说明性文件（如头文件.h文件、.inc文件、.def文件、编译说明文件.cfg等）头部应进行注释，注释必须列出：版权说明、版本号、生成日期、作者、内容、功能、与其它文件的关系、修改日志等，头文件的注释中还应有函数功能简要说明。

示例：下面这段头文件的头注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```
/************************************************
  Copyright (C), 1988-1999, Huawei Tech. Co., Ltd.
  File name:      // 文件名
  Author:      Version:       Date: // 作者、版本及完成日期
  Description:    // 用于详细说明此程序文件完成的主要功能，与其他模块
                  // 或函数的接口，输出值、取值范围、含义及参数间的控
                  // 制、顺序、独立或依赖等关系
  Others:         // 其它内容的说明
  Function List:  // 主要函数列表，每条记录应包括函数名及功能简要说明
    1. ....
  History:        // 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
    1. Date:
       Author:
       Modification:
    2. ...
************************************************/
```

[1]2-3：源文件头部应进行注释，列出：版权说明、版本号、生成日期、作者、模块目的/功能、主要函数及其功能、修改日志等。

示例：下面这段源文件的头注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```
/************************************************************
  Copyright (C), 1988-1999, Huawei Tech. Co., Ltd.
  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。

[1]2-4：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、调用关系（函数、表）等。

示例：下面这段函数的注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```
/*************************************************
  Function:       // 函数名称
  Description:    // 函数功能、性能等的描述
  Calls:          // 被本函数调用的函数清单
  Called By:      // 调用本函数的函数清单
  Table Accessed: // 被访问的表（此项仅对于牵扯到数据库操作的程序）
  Table Updated:  // 被修改的表（此项仅对于牵扯到数据库操作的程序）
  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
*************************************************/
```

[1]2-5：边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。

[1]2-6：注释的内容要清楚、明了，含义准确，防止注释二义性。

说明：错误的注释不但无益反而有害。

规则2-7：避免在注释中使用缩写，特别是非常用缩写。

说明：在使用缩写时或之前，应对缩写进行必要的说明。

[1]2-8：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。

示例：如下例子不符合规范。

例1：

```
/* get replicate sub system index and net indicator */


repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

例2：

```
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
/* get replicate sub system index and net indicator */
```

应如下书写

```
/* get replicate sub system index and net indicator */
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

[1]2-9：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。变量、常量、宏的注释应放在其上方相邻位置或右方。

示例：

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

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

[1]2-10：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*/
};
```

[1]2-11：全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它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 — no use  */      // 变量取值范围
/* only  function  SCCPTranslate() in */
/* this modual can modify it,  and  other */
/* module can visit it through call */
/* the  function GetGTTransErrorCode() */    // 使用方法
BYTE g_GTTranErrorCode;
```

[1]2-12：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。

示例：如下例子，排版不整齐，阅读稍感不方便。

```
void example_fun( void )
{
/* code one comments */
   CodeBlock One
```

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

应改为如下布局。

```
void example_fun( void )
{
    /* code one comments */
    CodeBlock One

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

[1]2-13：将注释与其上面的代码用空行隔开。

示例：如下例子，显得代码过于紧凑。

```
/* code one comments */
program code one
/* code two comments */
program code two
```

应如下书写

```
/* code one comments */
program code one

/* code two comments */
program code two
```

[1]2-14：对变量的定义和分支语句（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。

[1]2-15：对于switch语句下的case语句，如果因为特殊情况需要处理完一个case后进入下一个case处理，必须在该case语句处理完、下一个case语句前加上明确的注释。

说明：这样比较清楚程序编写者的意图，有效防止无故遗漏break语句。

示例（注意斜体加粗部分）：

```
case CMD_UP:
    ProcessUp();
    break;

case CMD_DOWN:
    ProcessDown();
    break;

case CMD_FWD:
    ProcessFwd();

if (...)
{
    ...
    break;
}
else
{
    ProcessCFW_B();   // now jump into case CMD_A
}

case CMD_A:
    ProcessA();
    break;

case CMD_B:
    ProcessB();
    break;

case CMD_C:
    ProcessC();
     break;

case CMD_D:
    ProcessD();
```

```
    break;
...
```

½2-1：避免在一行代码或表达式的中间插入注释。

说明：除非必要，不应在代码或表达中间插入注释，否则容易使代码可理解性变差。

½2-2：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的结构，使代码成为自注释的。

说明：清晰准确的函数、变量等的命名，可增加代码可读性，并减少不必要的注释。

½2-3：在代码的功能、意图层次上进行注释，提供有用、额外的信息。

说明：注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。

示例：如下注释意义不大。

```
/* if receive_flag is TRUE */
if (receive_flag)
```

而如下的注释则给出了额外有用的信息。

```
/* if mtp receive a message from links */
if (receive_flag)
```

½2-4：在程序块的结束行右方加注释标记，以表明某程序块的结束。

说明：当代码段较长，特别是多重嵌套时，这样做可以使代码更清晰，更便于阅读。

示例：参见如下例子。

```
if (...)
{
    // program code

    while (index < MAX_INDEX)
    {
        // program code
    } /* end of while (index < MAX_INDEX) */ // 指明该条while语句结束
} /* end of  if (...)*/ // 指明是哪条if语句结束
```

½2-5：注释格式尽量统一，建议使用“/* …… */”。

½2-6：注释应考虑程序易读及外观排版的因素，使用的语言若是中、英兼有的，建议多使用中文，除非能用非常流利准确的英文表达。

说明：注释语言不统一，影响程序易读性和外观排版，出于对维护人员的考虑，建议使用中文。

3 标识符命名

[1]3-1：标识符的命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。

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 ;

[1]3-2：命名中若使用特殊约定或缩写，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。

[1]3-3：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）。

[1]3-4：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。

示例：下面所示的局部变量名的定义方法可以借鉴。

int liv_Width

其变量名解释如下：

l 局部变量（Local） （其它：g 全局变量（Global）...）

i 数据类型（Interger）

v 变量（Variable） （其它：c 常量（Const）...）

Width 变量含义

这样可以防止局部变量与全局变量重名。

¹3-5：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，比如采用UNIX的全小写加下划线的风格或大小写混排的方式，不要使用大小写与下划线混排的方式，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。

½3-1：除非必要，不要用数字或较奇怪的字符来定义标识符。

示例：如下命名，使人产生疑惑。

```
#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 );
```

½3-2：在同一软件产品内，应规划好接口部分标识符（变量、结构、函数及常量）的命名，防止编译、链接时产生冲突。

说明：对接口部分的标识符应该有更严格限制，防止冲突。如可规定接口部分的变量与常量之前加上“模块”标识等。

½3-3：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
```

示例：

```
int  min_sum;
```

```
int  max_sum;
int  add_user( BYTE *user_name );
int  delete_user( BYTE *user_name );
```

½3-4：除了编译开关/头文件等特殊应用，应避免使用_EXAMPLE_TEST_之类以下划线开始和结尾的定义。

4 可读性

[1]4-1：注意运算符的优先级，并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。

[1]4-2：避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的枚举或宏来代替。

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。

```
#define TRUNK_IDLE 0
#define TRUNK_BUSY 1

if (Trunk[index].trunk_state == TRUNK_IDLE)
{
```

```
    Trunk[index].trunk_state = TRUNK_BUSY;
    ...  // program code
}
```

½4-1：源程序中关系较为紧密的代码应尽可能相邻。

说明：便于程序阅读和查找。

示例：以下代码布局不太合理。

```
rect.length = 10;
char_poi = str;
rect.width = 5;
```

若按如下形式书写，可能更清晰一些。

```
rect.length = 10;
rect.width = 5; // 矩形的长与宽关系较密切，放在一起。
char_poi = str;
```

½4-2：不要使用难懂的技巧性很高的语句，除非很有必要时。

说明：高技巧语句不等于高效率的程序，实际上程序的效率关键在于算法。

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; ”
```

5 变量、结构

[1]5-1：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。

[1]5-2：仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。

说明：在对变量声明的同时，应对其含义、作用及取值范围进行注释说明，同时若有必要还应说明与其它变量的关系。

[1]5-3：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。

说明：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。

示例：在源文件中，可按如下注释形式说明。

```
RELATION    System_Init    Input_Rec    Print_Rec   Stat_Score
Student     Create         Modify       Access      Access
Score       Create         Modify       Access      Access, Modify
```

注：RELATION为操作关系；System_Init、Input_Rec、Print_Rec、Stat_Score为四个不同的函数；Student、Score为两个全局变量；Create表示创建，Modify表示修改，Access表示访问。

其中，函数Input_Rec、Stat_Score都可修改变量Score，故此变量将引起函数间较大的耦合，并可能增加代码测试、维护的难度。

[1]5-4：当向公共变量传递数据时，要十分小心，防止赋与不合理的值或越界等现象发生。

说明：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。

[1]5-5：防止局部变量与公共变量同名。

说明：若使用了较好的命名规则，那么此问题可自动消除。

[1]5-6：严禁使用未经初始化的变量作为右值。

说明：特别是在C/C++中引用未经赋值的指针，经常会引起系统崩溃。

½5-1：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。

说明：降低公共变量耦合度。

½5-2：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：使用标准的数据类型，有利于程序的移植。

示例：如下例子（在DOS下BC3.1环境中），在移植时可能产生问题。

```
void main()
{
    register int index; // 寄存器变量

    _AX = 0x4000; // _AX是BC3.1提供的寄存器“伪变量”
    ... // program code
}
```

½5-3：结构的功能要单一，是针对一种事务的抽象。

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。

示例：如下结构不太清晰、合理。

```
typedef struct STUDENT_STRU
{
    unsigned char name[8]; /* student's name */
    unsigned char age;    /* student's age */
    unsigned char sex;    /* student's sex, as follows */
                        /* 0 - FEMALE; 1 - MALE */
    unsigned char
          teacher_name[8]; /* the student teacher's name */
    unisgned char
          teacher_sex;    /* his teacher sex */
} STUDENT;
```

若改为如下，可能更合理些。

```
typedef struct TEACHER_STRU
{
   unsigned char name[8]; /* teacher name */
   unisgned char sex;    /* teacher sex, as follows */
                      /* 0 - FEMALE; 1 - MALE */
} TEACHER;

typedef struct STUDENT_STRU
{
   unsigned char name[8];    /* student's name */
   unsigned char age;        /* student's age */
   unsigned char sex;        /* student's sex, as follows */
                         /* 0 - FEMALE; 1 - MALE */
   unsigned int  teacher_ind; /* his teacher index */
} STUDENT;
```

½5-4：不要设计面面俱到、非常灵活的数据结构。

说明：面面俱到、灵活的数据结构反而容易引起误解和操作困难。

½5-5：不同结构间的关系不要过于复杂。

说明：若两个结构间关系较复杂、密切，那么应合为一个结构。

示例：如下两个结构的构造不合理。

```
typedef struct PERSON_ONE_STRU
{
   unsigned char name[8];
   unsigned char addr[40];
   unsigned char sex;
   unsigned char city[15];
} PERSON_ONE;

typedef struct PERSON_TWO_STRU
{
   unsigned char name[8];
   unsigned char age;
   unsigned char tel;
} PERSON_TWO;
```

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。

```
typedef struct PERSON_STRU
{
    unsigned char name[8];
    unsigned char age;
    unsigned char sex;
    unsigned char addr[40];
    unsigned char city[15];
    unsigned char tel;
} PERSON;
```

½5-6：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。

示例：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 PERSON_BASE_INFO_STRU
{
    unsigned char name[8];
    unsigned char age;
    unsigned char sex;
} PERSON_BASE_INFO;

typedef struct PERSON_ADDRESS_STRU
{
    unsigned char addr[40];
    unsigned char city[15];
    unsigned char tel;
} PERSON_ADDRESS;

typedef struct PERSON_STRU
{
    PERSON_BASE_INFO person_base;
    PERSON_ADDRESS person_addr;
} PERSON;
```

½5-7：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;
```

½5-8：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。

½5-9：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。

½5-10：编程时，要注意数据类型的强制转换。

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。

½5-11：对编译系统默认的数据类型转换，也要有充分的认识。

示例：如下赋值，多数编译器不产生告警，但值的含义还是稍有变化。

```
char chr;
unsigned short int exam;

chr = -1;
exam = chr; // 编译器不产生告警，此时exam为0xFFFF。
```

½5-12：尽量减少没有必要的数据类型默认转换与强制转换。

½5-13：合理地设计数据并使用自定义数据类型，避免数据间进行不必要的类型转换。

½5-14：对自定义数据类型进行恰当命名，使它成为自描述性的，以提高代码可读性。注意其命名方式在同一产品中的统一。

说明：使用自定义类型，可以弥补编程语言提供类型少、信息量不足的缺点，并能使程序清晰、简洁。

示例：可参考如下方式声明自定义数据类型。

下面的声明可使数据类型的使用简洁、明了。

```
typedef unsigned char  BYTE;
typedef unsigned short WORD;
typedef unsigned int   DWORD;
```

下面的声明可使数据类型具有更丰富的含义。

```
typedef float DISTANCE;
typedef float SCORE;
```

½5-15：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：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。

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

EXAM_BIT A1 A2 A3

说明：在对齐方式下，CPU的运行效率要快得多。

示例：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。

```
1       8       16      24      32
------- ------- ------- -------
| long1 | long1 | long1 | long1 |
------- ------- ------- -------
|       |       |       | long2 |
------- ------- ------- --------
| long2 | long2 | long2 |       |
------- ------- ------- --------
| ....
```

11 代码测试、维护