1. C编码规范

1. C编码规范

1.1. 排版规范

规则-1：对齐和缩进，必须使用4个空格为缩进单位，不可以使用tab。

规则-2：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。

规则-3：相对独立的程序块之间、变量说明之后必须加空行。

规则-4：较长的语句（约80字符左右）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。

规则-5：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。

规则-6：不允许把多个短语句写在一行中，即一行只写一条语句。

规则-7：左花括号｛的书写位置，可以单独一行或紧跟在if,else等后面，右花括号｝必须单独一行。无论使用哪种，必须保证源代码中一致。

规则-8：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{}。

规则-9：在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如－>），后不应加空格。

1.2. 命名规范

规则-1：变量和函数名必须与所属项目风格一致。

规则-2：无用的变量和函数，必须删掉。

规则-3：定义非static限定函数，名称必须加上所属项目前缀。防止与其他库发生冲突。

规则-4：如非必要，不要定义全局变量。对于必须定义的全局变量，名称必须加上所属系统+项目前缀。

规则-5：使用特殊约定或缩写，则要有注释说明。

规则-6：除临时循环变量外，禁止取单个字符(如a,b,i,j...)作为变量名称。

规则-7：非特殊情况，不允许使用数字或者较奇怪字符定义标识符。

规则-8：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。

1.3. 日志规范

规则-1：精简正常流程的日志输出，尽量减少在循环中打印日志。

规则-2：异常流程中，必须有有效日志输出。

规则-3：如非客户特殊要求，日志的内容必须为中文，且语句必须表达通顺。不允许在已提交的程序中发现乱写的无效日志和跟踪日志。

规则-4：对于提供了标准日志分级的系统，必须要严格遵守日志分级的策略。

1.4. 逻辑规范

规则-1：除底层算法类程序，业务处理程序外，不可以使用超过3次以上（含3次）循环。

规则-2：除守护进程外，不允许编写死循环。循环次数超过1000次以上，要有必要的分段跟踪日志。

1.5. 函数规范

规则-1：防止将函数的参数作为工作变量。

规则-2：函数的规模尽量限制在200行以内。

规则-3：一个函数仅完成一件功能。

规则-4：为简单功能编写函数。

规则-5：不要设计多用途面面俱到的函数。

规则-6：函数的功能应该为可预测的，也就是输入同样的参数，就应产生同样的输出。

规则-7：避免设计多参数函数，不使用的参数从接口中去掉。

规则-8：定义函数时，必须明确返回值类型，不可以使用系统默认类型。

1.6. 可靠性规范

规则-1：代码质量保证优先原则。

规则-2：禁止数组下表越界，禁止内存操作越界。

规则-3：防止空指针操作。

规则-4：声明的变量需要进行初始化，特别禁止对该变量还有类似+=等累加操作的。

规则-5：防止给变参函数错误的参数匹配。

规则-6：防止野指针。指针所指向的空间被释放后，必须将该指针置为NULL;

规则-7：使用open,fopen,pipe,socket等打开系统资源的函数，必须在函数返回的时候，调用对应的释放函数进行资源释放。 同样适用于封装过的该类函数。

规则-8：防止内存泄漏，使用malloc,calloc,strdup等函数，必须在内存使用完毕后进行释放。 同样适用于封装过的该类函数。

规则-9：防止使用sizeof对返回值的指针进行计算。

规则-10：检查函数所有参数输入和非参数输入(数据文件、公共变量)的有效性。

1.7. 性能规范

规则-1：不允许为了未知的扩展性，浪费大量程序空间。

规则-2：可以不在循环中执行的重复动作，一定要提取出去。

规则-3：如非必要，不允许在循环中动态申请资源。尽量减少循环中的消耗。

规则-4：查询数据库时，尽量使用索引，且查询条件的顺序必须与复合索引的顺序一致。

规则-5：查询数据库时，对于数据库的类型，必须正确的使用‘’。

1.1. 排版规范

规则-1：对齐和缩进，必须使用4个空格为缩进单位，不可以使用tab。

说明：对于由开发工具自动生成的代码可以有不一致。

规则-2：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。

规则-3：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;
```

规则-4：较长的语句（约80字符左右）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：

```
sprintf( wherelist, " txbrno='%s' and platdate=%ld and traceno=%ld and sts='1'",

hvps_rcvlist.txbrno, hvps_rcvlist.platdate, hvps_rcvlist.traceno);

act_task_table(taskno).duration_true_or_false

 = SYS_get_sccp_statistic_state( stat_item );
```

规则-5：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
}
```

规则-6：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：使用逗号定义变量是允许的（int i=0,j=0,k=0;）。

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;
```

规则-7：左花括号｛的书写位置，可以单独一行或紧跟在if,else等后面，右花括号｝必须单独一行。无论使用哪种，必须保证源代码中一致。

示例：

```
if( ret != 0)

{

APPLOG("E","调用签名服务器失败(%d)!!, ret");

return -5;

}
```

或

```
if( ret != 0){

APPLOG("E","调用签名服务器失败(%d)!!, ret");

 return -5;

}
```

规则-8：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;

}
```

规则-9：在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如－>），后不应加空格。

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。

1.2. 命名规范

规则-1：变量和函数名必须与所属项目风格一致。

说明：在一致的基础上，推荐使用匈牙利命名法。

规则-2：无用的变量和函数，必须删掉。

规则-3：定义非static限定函数，名称必须加上所属项目前缀。防止与其他库发生冲突。

说明：static限定的函数，不被约束。

示例：以下为错误的示例：

```
char * strtrim(char * string)

{

…

}
```

正确的示例:

```
static char * strtrim( char * string); /*仅本程序有效*/

char * xippmts_strtrim( char * string);/*二代支付系统专用*/
```

规则-4：如非必要，不要定义全局变量。对于必须定义的全局变量，名称必须加上所属系统+项目前缀。

规则-5：使用特殊约定或缩写，则要有注释说明。

规则-6：除临时循环变量外，禁止取单个字符(如a,b,i,j…)作为变量名称。

规则-7：非特殊情况，不允许使用数字或者较奇怪字符定义标识符。

示例：错误写法：

```
double amt1=0.00;/*交易金额*/

double amt2=0.00;/*手续费*/

double amt3=0.00;/*服务费*/
```

正确写法：

```
double tx_amt=0.00;

double handle_amt=0.00;

double service_amt=0.00;
```

规则-8：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 );
```

1.3. 日志规范

规则-1：精简正常流程的日志输出，尽量减少在循环中打印日志。

规则-2：异常流程中，必须有有效日志输出。

规则-3：如非客户特殊要求，日志的内容必须为中文，且语句必须表达通顺。不允许在已提交的程序中发现乱写的无效日志和跟踪日志。

示例：以下注释不允许在已提交的程序中被发现：

```
APPLOG(“D”,”SSSSSSSSSSSSSSSSSS”);

APPLOG(“E”,”(%s)”, tmpstr);

APPLOG(“E”,”hahahaha(%d)”,ret);
```

以下为正确的日志内容格式：

```
APPLOG("E","执行表hvps_rcvlist更新游标错误(%d)(%s)",ret,wherelist);

APPLOG("D","收到一代 (%s)(%s)",

hvps_rcvlist.cmtno,hvps_rcvlist.msgid);
```

规则-4：对于提供了标准日志分级的系统，必须要严格遵守日志分级的策略。

1.4. 逻辑规范

规则-1：除底层算法类程序，业务处理程序外，不可以使用超过3次以上（含3次）循环。

规则-2：除守护进程外，不允许编写死循环。循环次数超过1000次以上，要有必要的分段跟踪日志。

1.5. 函数规范

规则-1：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。

示例：下函数的实现不太好。

```
void sum_data( unsigned int num, int *data, int *sum )

{

 unsigned int count;



 *sum = 0;

 for (count = 0; count < num; count++)

 {

 *sum += data(count); // sum成了工作变量，不太好。

 }

}
```

若改为如下，则更好些：

```
void sum_data( unsigned int num, int *data, int *sum )

{

 unsigned int count ;

 int sum_temp;



 sum_temp = 0;

 for (count = 0; count < num; count ++)

 {
```

```
 sum_temp += data(count);

 }



 *sum = sum_temp;

}
```

规则-2：函数的规模尽量限制在200行以内。

规则-3：一个函数仅完成一件功能。

规则-4：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。

规则-5：不要设计多用途面面俱到的函数。

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。

规则-6：函数的功能应该为可预测的，也就是输入同样的参数，就应产生同样的输出。

说明：如非必要，减少static等函数内部存储标记。如非必要，不可使用random等产生返回。

规则-7：避免设计多参数函数，不使用的参数从接口中去掉。

规则-8：定义函数时，必须明确返回值类型，不可以使用系统默认类型。

示例：错误的写法：

```
spB001()

{

…

}
```

正确的写法：

```
int spB001()

{
```

…

}

1.6. 可靠性规范

规则-1：代码质量保证优先原则。

\1. 正确性，指程序要实现设计要求的功能。

\2. 稳定性、安全性，指程序稳定、可靠、安全。

\3. 可测试性，指程序要具有良好的可测试性。

\4. 规范/可读性，指程序书写风格、命名规则等要符合规范。

\5. 全局效率，指软件系统的整体效率。

\6. 局部效率，指某个模块/子模块/函数的本身效率。

\7. 个人表达方式/个人方便性，指个人编程习惯。

规则-2：禁止数组下表越界，禁止内存操作越界。

示例：错误1，数组定义时，未考虑'\0'结束位：

char tx_brno(5);

strcpy(tx_brno, "88888");

错误2,将大数组赋值给小数组：

char filename(20);

char tmpstr(50);

strcpy(filename,tmpstr);

规则-3：防止空指针操作。

说明：对于指针，一定要防止对空指针进行操作。

规则-4：声明的变量需要进行初始化，特别禁止对该变量还有类似+=等累加操作的。

规则-5：防止给变参函数错误的参数匹配。

示例：错如的代码如下：

```
printf("文件不存在(%s)\n");

printf("文件不存在(%s)\n",ret,filename);
```

规则-6：防止野指针。指针所指向的空间被释放后，必须将该指针置为NULL；

示例：

```
char * recvBuf=NULL;

recvBuf = (char *)malloc(1000);

do something….

free(recvBuf);

recvBuf=NULL;
```

规则-7：使用open,fopen,pipe,socket等打开系统资源的函数，必须在函数返回的时候，调用对应的释放函数进行资源释放。 同样适用于封装过的该类函数。

规则-8：防止内存泄漏，使用malloc,calloc,strdup等函数，必须在内存使用完毕后进行释放。 同样适用于封装过的该类函数。

规则-9：防止使用sizeof对返回值的指针进行计算。

示例：如下是错误的使用：

```
char * a = NULL;

a = (char *)malloc(filelen+1);

memset( a, 0x00, sizeof(a));
```

正确的写法应该是：

```
memset( a, 0x00, sizeof(filelen+1));
```

规则-10：检查函数所有参数输入和非参数输入(数据文件、公共变量)的有效性。

1.7. 性能规范

编程时要经常注意代码的效率。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。

规则-1：不允许为了未知的扩展性，浪费大量程序空间。

说明：如果对长度等具有不确定性，请使用malloc等进行动态申请。

以下示例为错误的：

```
char wherelist(102400);
```

规则-2：可以不在循环中执行的重复动作，一定要提取出去。

示例：如下代码为错误的：

```
while(1)

{

get_zd_long(“0440”, &plat_date);

 do something….

}
```

规则-3：如非必要，不允许在循环中动态申请资源。尽量减少循环中的消耗。

规则-4：查询数据库时，尽量使用索引，且查询条件的顺序必须与复合索引的顺序一致。

规则-5：查询数据库时，对于数据库的类型，必须正确的使用``。

示例：错误的写法如下：

```
sprintf( wherelist, “platdate=‘%08ld’ and plattraceno=%08ld and txbrno=%s”,

 platdate, plattraceno, txbrno);
```

正确的写法应该是：

```
sprintf( wherelist, “platdate=%08ld and plattraceno=%08ld and txbrno=‘%s’”,
```

platdate, plattraceno, txbrno);