中华人民共和国国家军用标准

FL 0137　　GJB 5369—2005

航天型号软件 C 语言安全子集

Safe subset of C language for space armament software

2005-04-11 发布　　2005-07-01 实施

国防科学技术工业委员会　发布

目　　次

前　言

本标准是参照 MISRA (Motor Industry Software Reliability Association) 1998 年的《Guidelines For The Use Of The C Language In Vehicle Based Software》和 LDRA (Liverpool Data research Associates) 2000 年的《MISRA C Checking》，并结合航天型号软件的特点经过裁剪和补充而形成的。

本标准的附录 A 为资料性附录。

本标准由航天科工集团公司提出。

本标准由中国航天标准化研究所归口。

本标准起草单位：航天科工集团公司二院 706 所。

本标准主要起草人：宋晓秋。

航天型号软件C语言安全子集

1　范围

本标准规定了C语言软件的编程准则。准则分为推荐和强制两种类型，推荐类为参照执行的准则，强制类为必须执行的准则。

本标准适用于航天型号C语言软件。

2　规范性引用文件

下列文件中的有关条款通过引用而成为本标准的条款。凡注日期或版次的引用文件，其后的任何修改单(不包括勘误的内容)或修订版本都不适用于本标准，但提倡使用本标准的各方探讨使用其最新版本的可能性。凡不注日期或版次的引用文件，其最新版本适用于本标准。

GB/T 11457－1995　软件工程术语

3　术语和定义

GB/T 11457－1995 确定的术语和定义适用于本标准。

4　细则

4.1　声明定义类

4.1.1　强制类

4.1.1.1　过程名禁止被重用

一个过程名禁止被重用于其它之处。

例如：

```
Void foo(unsigned int p_1)
{
    unsigned int x=p_1;
}
/*****************************************
* 过程名禁止被重用
*****************************************/
void static_p(void)
{
    unsigned int foo=1u;
}
```

4.1.1.2　标号名禁止被重用

一个标号名被重用掩盖了这个标号名的含义。出现这种情况时，编译是否通过是依赖于编译器的差异而不同。

例如：

```
/*********************************************************************
* 标号名禁止被重用
*********************************************************************/
void static_p(void)
{
```

```
    unsigned int value_x=lu;
    unsigned int y=0u;
    /*...*/
value_x:
    y=1u;
    /*...*/
}
```

4.1.1.3 禁止在结构体定义中含有空域

在结构体定义中禁止含有空域。该准则的违背通常是出现在类似于如下的使用中：

struct atag{struct anothertag{...}； ...}；

例如：

```
/*****************************************************************
 * 禁止在结构体定义中含有空域
 *****************************************************************/
struct s_p {unsigned int xs； struct {unsigned char ac， ab； }； }；
void static_p(void)
{
    struct s_p sb;
    sb.xs=1；
    /*...*/
}
```

4.1.1.4 禁止声明多重标号

使用多重标号是多余的，多重标号完全可以用一个标号来替代。

例如：

```
/*************************************************************
 * 禁止声明多重标号
 *************************************************************/
static void static_p(void)
{
start： begin：
    /*...*/;
}
```

4.1.1.5 参数必须使用类型声明

虽然一些编译器允许缺省参数类型，但使用参数类型说明易于类型匹配的检查，因此参数必须使用类型声明。

例如：

```
/*********************************************************************
 * 参数必须使用类型声明
 *********************************************************************/
unsigned int static_p(p_1)
{
    unsigned int result;
    /*...*/
    result=p_1*2;
    return result;
}
```

4.1.1.6 在过程声明中必须对参数说明

虽然大多数编译器允许在过程声明中省略任何的参数说明，但在过程声明中对参数进行说明易于对过程使用中参数类型匹配的检查，因此在过程声明中必须对参数说明。

例如：

```
/*****************************************************************
 * 在过程声明中必须对参数说明
 *****************************************************************/
int static_p( );
int static_p(unsigned int p)
{
    int x=1;
    /*...*/
    if (p= =0)
    {
      x=0;
    }
    return x;
}
```

4.1.1.7 禁止过程参数只有类型没有标识符

一些编译器允许过程参数的说明只有类型而没有标识符，但这样的参数并不能真正地被使用，因此禁止过程参数只有类型没有标识符。

例如：

```
Struct s_type_b{unsigned int xs; };
/*****************************************************************
 * 禁止过程参数只有类型没有标识符
 *****************************************************************/
void static_p_a(unsigned int p_1， struct s_type_b *);
void static_p_a(unsigned int p_1， struct s_type_b *)
{
    /*...*/
}
```

4.1.1.8 禁止在过程参数表中使用省略号

过程参数用省略号说明不利于对参数匹配的分析，因此禁止在过程参数表中使用省略号。

例如：

```
/*****************************************************************
 * 禁止在过程参数表中使用省略号
 *****************************************************************/
unsigned int static_p(unsigned char* p_1， ...)
{
    /*...*/
    return 1u;
}
```

4.1.1.9 禁止重新定义使用 C 或 C++的关键字

重新定义使用 C 或 C++的关键字，破坏了程序的可读性，因此禁止重新定义使用 C 或 C++的关键字。

例如：

```
/*********************************************
 * 禁止重新定义使用 C 或 C++关键字
 *********************************************/
void static_p(void)
{
    unsigned int public    =0;
    unsigned int private   =0;
    unsigned int protected =0;
    unsigned int operator  =0;
    unsigned int new       =0;
    unsigned int template  =0;
    unsigned int virtual   =0;
    unsigned int delete    =0;
    unsigned int friend    =0;
    unsigned int cout      =0;
    unsigned int cin       =0;
    unsigned int endl      =0;
    /*...*/
}
```

4.1.1.10 禁止过程或函数中的参数表为空

如果是一个无参数过程或函数，必须使用 func(void)的形式说明，禁止使用 func()的形式说明。

例如：

```
/*****************************************************************************
 * 禁止过程或函数中的参数表为空
 *****************************************************************************/
void satic_p( )
{
    /*...*/
}
```

4.1.1.11 禁止在同一个宏中使用多个#或##

在同一个宏中使用多于一个的#或##，或同时使用#和##都是很危险的，因此禁止在同一个宏中使用多个#或##。

4.1.1.12 禁止定义不象函数的宏

定义带参数的宏(类函数宏)时，宏体必须用括号括起来。

例如：

```
#define IF_X(x)  if(x)  {
/*****************************************************************
 * 禁止定义不象函数的宏
 *****************************************************************/
void static_p(void)
{
    bool test=true;
    IF_X(test)
        test=!test;
    }
}
```

4.1.1.13 禁止在宏中包含不允许的项

宏只能用于符号常量，类函数宏，类型限定符以及存储类说明。宏中不允许含有语句关键字和类型关键字。

例如：

```
/*************************************************************
 * 禁止在宏中包含不允许的项
 *************************************************************/
#define static_p unsigned int
void test_p(void)
{
    static_p x=1u;
    /*...*/
}
```

4.1.1.14 禁止重新定义保留字

宏预处理可以重新定义保留字，但这种做法会引起混淆，因此禁止重新定义保留字。

例如：

```
/*************************************************************
 * 禁止重新定义保留字
 *************************************************************/
#define FILE unsigned int
void dummy(void)
{
    /*...*/
}
```

4.1.1.15 字符型变量必须明确定义是有符号还是无符号

若对字符型变量不明确定义是否有符号，则在执行时会作为有符号或无符号的类型使用，因此要求字符型变量必须明确定义是有符号还是无符号。

例如：

```
/*************************************************************
 * 字符型变量必须明确定义是有符号还是无符号
 *************************************************************/
void static_p(void)
{
    char c='c';
    /*...*/
}
```

4.1.1.16 禁止对一个名字重新定义

虽然 C 语言允许在许多不同的上下文中对一个名字重新定义，但命名的唯一性可使程序增加可读性，因此禁止对一个名字重新定义。

例如：

```
/*************************************************************
 * 禁止对一个名字重新定义
 *************************************************************/
unsigned int static_p(void);
struct static_p
{
```

```
    unsigned int static_p;
    unsigned int u_1;
};
unsigned int static_p(void)
{
    unsigned int var_1;
    /*...*/
static_p:
    var_1=1u;
    /*...*/
    return(var_1);
}
```

4.1.1.17 用 typedef 自定义的类型禁止被重新定义

改变用户自定义的类型会引起混淆甚至能导致错误，因此用 typedef 自定义的类型禁止被重新定义。

例如：

```
typedef int mytype;
/*****************************************************************************
 * 用 typedef 自定义的类型禁止被重新定义
 *****************************************************************************/
void static_p(void)
{
    typedef float mytype;
    /*...*/
}
```

4.1.1.18 禁止在同一个文件中有#if 而没有#endif

预处理语句的#if 和#endif 禁止分散在不同的文件之中。

4.1.1.19 禁止数组没有边界限定

禁止使用没有边界限定的数组定义。

例如：

```
/************************************************************
 * 禁止数组没有边界限定
 ************************************************************/
void static_p(void)
{
  unsigned int u_array[]={0，1，2};
  /*...*/
}
```

4.1.1.20 禁止在#include<……>中使用绝对路径名

头文件路径应该在编译器的选项中予以设置说明，禁止在#include<……>中使用绝对路径名。

例如：

```
/************************************************************
 * 禁止在#include<……>中使用绝对路径名
 ************************************************************/
#include <C：\VC\include\stdio.h>
void Dummy(void)
{
```

```
    /*...*/
}
```

4.1.1.21 禁止结构体声明不完整

结构体的声明必须要完整地声明。

例如：

```
/************************************************************
 * 禁止结构体声明不完整
 ************************************************************/
struct static_p;
void   dummy(void)
{
    /*...*/
}
```

4.1.1.22 禁止参数的声明形式上不一致

在参数表的参数声明中，对所有参数的类型和变量名的声明形式上必须保持一致。推荐使用仅对参数类型进行声明的形式。

例如：

```
/************************************************************************
 * 禁止参数的声明形式上不一致
 ************************************************************************/
float static_p(unsigned int， unsigned short p_2)；
```

4.1.2 推荐类

4.1.2.1 建议使用 typedef 在统一的变量声明头文件中对基本变量类型重新定义

在不同的编译程序中基本类型的长度是不相同的，因此为了代码易于移植，推荐在统一的变量声明头文件中使用 typedef 定义软件使用的变量类型，而在除此之外的程序中不再使用基本类型的声明。

例如，在头文件 c_standards.h 中定义：

```
typedef unsigned int      UINT_32；
typedef int               SINT_32；
typedef unsigned short    UINT_16；
typedef unsigned char     UCHAR；
typedef float             FLOAT_32；
typedef double            FLOAT_64；
```

在程序中的使用如下：

```
#include " c_standards.h"
void p(void)
{
    UINT_32 i=0u；
    FLOAT_32 x=0.0f；
    ……
}
```

4.1.2.2 避免将过程定义为参数

将过程定义为参数可以用来生成代码，但也会使程序的控制流变得不清晰，因此避免将过程定义为参数。

例如：

```
Void test_p(int p_1)
{/*...*/}
/************************************************************
 * 避免将过程定义为参数
 ************************************************************/
void static_p(void(*p_proc_pointer)(int parameter_1))
{
    p_proc_pointer=test_p;
}
```

4.1.2.3 过程中避免使用过多的参数，建议不要超过 20 个

使用全局变量与使用参数传递之间的权衡是一个复杂的问题。通过实参与形参传递数据确实会消耗一些时间，而使用全局变量会快一些。但过多地使用全局变量也会带来许多副作用。减少参数个数的方法之一是可以将相关参数定义在结构体中，这样使过程看起来更简洁。一般建议通过实参与形参传递数据的参数个数不要超过 20 个。

例如：

```
/************************************************************
 * 过程中避免使用过多的参数，建议不要超过 20 个
 ************************************************************/
void static_p(unsigned int p_1,  unsigned int p_2,  unsigned int p_3,
              unsigned int p_4,  unsigned int p_5,  unsigned int p_6,
              unsigned int p_7,  unsigned int p_8,  unsigned int p_9,
              unsigned int p_10, unsigned int p_11, unsigned int p_12,
              unsigned int p_13, unsigned int p_14, unsigned int p_15,
              unsigned int p_16, unsigned int p_17, unsigned int p_18,
              unsigned int p_19, unsigned int p_20, unsigned int p_21)
{
    /*...*/
}
```

4.1.2.4 在结构体定义中谨慎使用位域

位域是 C 中的一个难点，因为对它的使用必须特别谨慎。

例如：

```
/************************************************************************
 * 在结构体定义中谨慎使用位域
 ************************************************************************/
struct bitfield1{unsigned int x:1; };
/*-- Function prototypes --*/
void dummy(void)
{
    /*...*/
}.
```

4.1.2.5 避免在一个程序块中单独使用#define

宏定义会使程序的可读性降低，尤其是那些定义在内部块中的宏使用起来是很危险的，因为块和块的作用域是不相关的，因此避免在一个程序块中单独使用#define。

例如：

```
/*****************************************************************
 * 避免在一个程序块中单独使用#define
 *****************************************************************/
void static_p(unsigned int p_1)
{
    unsigned int local_x=0u;
    /*...*/
#define BLOCKDEF 1u
    local_x=local_x+BLOCKDEF;
    /*...*/
}
```

4.1.2.6 避免在一个程序块中单独使用#undef

宏定义会使程序的可读性降低，尤其是那些定义在内部块中的宏使用起来是很危险的，因为块和块的作用域是不相关的，因此避免在一个程序块中单独使用#undef。

例如：

```
#define BLOCKDEF 1u
/*****************************************************************
 * 避免在一个程序块中单独使用#undef
 *****************************************************************/
void static_p(void)
{
    unsigned int local_x=0u;
    /*...*/
    local_x=local_x+BLOCKDEF;
#undef BLOCKDEF
    /*...*/
}
```

4.1.2.7 谨慎使用#pragma

#pragma 可以用来掩盖所有类型的问题，因此应谨慎使用#pragma。

例如：

```
/*****************************************
 * 谨慎使用#pragma
 *****************************************/
#pragma
void foo(void)
{
    /*...*/
}
```

4.1.2.8 谨慎使用联合(union)的声明

当覆盖范围不确定时，使用联合是很危险的，因此应谨慎使用联合(union)的声明。

例如：

```
/*********************************************
 * 谨慎使用联合(union)的声明
 *********************************************/
union static_p{float fu; unsigned int xu; };
void dummy(void)
```

```
{
    /*...*/
}
```

4.1.2.9 在结构体中谨慎使用无名位域

无名位域的使用可能确实是编程者的意图，也可能是编程者的失误，因此在结构体中谨慎使用无名位域。

例如：

```
/*************************************************************************
 * 在结构体中谨慎使用无名位域
 *************************************************************************/
struct static_p{unsigned int x:2，:2；};
void dummy(void)
{
    /*...*/
}
```

4.2 版面书写类

4.2.1 强制类

4.2.1.1 过程体必须用大括号括起来

基于加强代码可读性、避免人为失误的目的，过程体必须用大括号括起来。

4.2.1.2 循环体必须用大括号括起来

基于加强代码可读性、避免人为失误的目的，循环体必须用大括号括起来。

例如：

```
/*****************************************************
 * 循环体必须用大括号括起来
 *****************************************************/
int static_p(int p_1)
{
    int j=10;
    int k=0;
    /*...*/
    for（k=0；k<10；k=k+1）j--；
    return j;
}
```

4.2.1.3 then/else 中的语句必须用大括号括起来

基于加强代码可读性、避免人为失误的目的，then/else 中的语句必须用大括号括起来。

例如：

```
/*********************************************************
 * then/else 中的语句必须用大括号括起来
 *********************************************************/
int static_p(int p_1，int p_2)
{
    int i=1；
    int j=2；
    /*...*/
    if (p_1>0) {
```

```
    i=i-1;
  } else
    i=i+1;

  if (p_2>0) {
    j=j+p_2;
  } else if (p_2<0) {
    j=j-p_2;
  } else{
    j=i;
  }
  return i;
}
```

4.2.1.4 逻辑表达式的连接必须使用括号

在含有逻辑操作符的表达式中使用括号可使运算顺序变得清晰，且不容易出错，因此逻辑表达式的连接必须使用括号。

例如：

```
/*****************************************************************
 * 逻辑表达式的连接必须使用括号
 *****************************************************************/
void static_p(void)
{
    bool flag=true;
    unsigned int y=0u，x=0u，z=1u;
    /*…*/
    if (x<0 || z+y!=0&&!flag)
    {
        flag=false;
    }
}
```

4.2.1.5 禁止在头文件前有可执行代码

头文件应放在文件开始的地方，应在任何可执行代码之前。

例如：

```
/*****************************************************************
 * 禁止在头文件前有可执行代码
 *****************************************************************/
void static_p(void)
{
#include " myfile.h"
    /*...*/
}
```

4.2.1.6 宏参数必须用括号括起来

宏体内使用的参数必须用括号括起来。

例如：

```
/*****************************************************************
 * 宏参数必须用括号括起来
 *****************************************************************/
#define static_p(x)  x>=0?x:-x
void test_p(void)
{
    unsigned int result;
    int a=6，b=5;
    /*...*/
    result=static_p(a-b);
    result=static_p(a)+1;
    /*...*/
}
```

4.2.1.7 嵌入汇编程序的过程必须是纯汇编程序

该准则是为了确保所有插入的代码是有界面定义的纯过程。

例如：

```
/*****************************************************************
 * 嵌入汇编程序的过程必须是纯汇编程序
 *****************************************************************/
void static_p(void)
{
    unsigned int x;
    x=0u;
    _asm
    {
        mov eax，x
    }
}
```

4.2.1.8 头文件名禁止使用“‘”、“\”和“/*”等字符

头文件名使用“‘”、“\”和“/*”等字符会带来隐含的冲突，因此头文件名禁止使用“‘”、“\”和“/*”等字符。

4.2.1.9 禁止字符串中单独使用“\”，字符串的终止必须使用“\0”

只用“\”并不能控制字符串的终止，应该用“\0”控制字符串的终止。

例如：

```
/*************************************************
 * 禁止字符串中单独使用“\”，字符串的终止必须使用“\0”
 *************************************************/
void static_p(void)
{
    unsigned char* str=(unsigned char*)″ string\
                        literal″ ;
    /*...*/
}
```

4.2.1.10 main 必须定义为 int main(void)或 int main(int，char*[])的形式

main 必须定义为 int main(void)或 int main(int，char*[])的形式，这有助于错误的确定和参数的输入。

例如：

```
/*************************************************************
 * main 必须定义为 int main(void)或 int main(int， char*[])的形式
 *************************************************************/
void main(void)
{
    /*...*/
}
```

4.2.2 推荐类

4.2.2.1 建议一个文件中的程序总行不超过 2000 行

过长的程序文件不利于管理和维护。建议一个文件中的程序总行不超过 2000 行。

4.2.2.2 建议一个过程或函数中的程序总行不超过 200 行

过程或函数中过长的代码行不利于维护和单元测试。建议一个过程或函数中的程序总行不超过 200 行。

4.3 分支控制类

4.3.1 强制类

4.3.1.1 禁止条件判别成立时相应分支无执行语句

条件判别成立时相应分支中无任何执行语句，可能是由于疏忽而遗漏掉了，或是有意避免布尔表达式不成立时的情况。为了防止由于疏忽造成的遗漏，因此禁止条件判别成立时相应分支无执行语句。

该准则的违背通常为下面几种形式：

a) if(...)else

b) if(...){ }else

c) if(...){; }else

例如：

```
/*********************************************************************
 * 禁止条件判别成立时相应分支无执行语句
 *********************************************************************/
void static_p(void)
{
    unsigned int value_x=1u;
    /*...*/
    if (value_x= =0u);
    /*...*/
    if (value_x= =0u) {
#if FALSE
    value_x=value_x+1u;
#endif
    }
    /*...*/
    if (value_x= =0u) {; }
}
```

4.3.1.2 在 if...else if 语句中必须使用 else 分支

在 if...else if 语句中为了表明已经考虑了所有情况，必须使用 else 分支。

例如：

```
/*************************************************************
 * 在 if...else if 语句中必须使用 else 分支
 *************************************************************/
void static_p(void)
{
    unsigned int x=2u;
    /*...*/
    if (x= =2u) {
      /*…*/
    } else if (x= =3u) {
      /*...*/
    }
}
```

4.3.1.3 禁止条件判别的 else 分支无可执行语句

else 分支中无可执行语句或是由于代码不完整造成的，或是有意表明 else 对应的可能性已经考虑到了。为了防止残留不完整的代码，因此禁止条件判别的 else 分支无可执行语句。

该准则的违背通常为下面几种形式：

a) else;

b) else{ }

c) else{; }

例如：

```
/*********************************************************************
 * 禁止条件判别的 else 分支无可执行语句
 *********************************************************************/
void static_p(void)
{
    unsigned int name_x=1u;
    /*...*/
    if (name_x= =0u)
    {/*...*/}
    else;
    /*...*/
    if (name_x= =0u)
    {/*...*/}
    else
    {
#if FALSE
    name_x=name_x+1u;
#endif
    }
    /*...*/
    if (name_x= =0u)
    {/*...*/}
    else
    {; }
}
```

4.3.1.4 在 switch 语句中必须有 default 语句

如果 switch 语句中缺省了 default 语句，当所有的 case 语句的表达式值都不匹配时，则会跳转到整个 switch 语句后的下一个语句执行。强制 default 语句的使用体现出已考虑了各种情况的编程思想。

例如：

```
/*************************************************************
 * 在 switch 语句中必须有 default 语句
 *************************************************************/
void static_p(int p_1)
{
    int i=0，j=0;
    /*...*/
    switch(p_1)
    {
        case 0:
            j=0;
            break;
        case 1:
            j=i;
            break;
    }
    /*...*/
}
```

4.3.1.5 禁止使用空 switch 语句

空 switch 语句不具备任何实际的操作内容，因此禁止使用空 switch 语句。

例如：

```
/*****************************************************
 * 禁止使用空 switch 语句
 *****************************************************/
void static_p(int p_1)
{
    int i=p_1;
    switch (i)
    {
    }
}
```

4.3.1.6 禁止 switch 语句中只包含 default 语句

如果 switch 语句中只包含 default 语句，则该 switch 语句的使用无任何实际价值，因此禁止 switch 语句中只包含 default 语句。

例如：

```
/*************************************************
 * 禁止 switch 语句中只包含 default 语句
 *************************************************/
void static_p(int p_1)
{
    int i=p_1;
    switch(i)
```

```
    {
    default:
    i++;
    }
}
```

4.3.1.7 禁止 switch 的 case 语句不是由 break 终止

如果某个 case 语句最后的 break 被省略，在执行完该 case 语句后，系统会继续执行下一个 case 语句。case 语句不是有 break 终止，有可能是编程者的粗心大意，也有可能是编程者的特意使用。为了避免编程者的粗心大意，因此禁止 switch 的 case 语句不是由 break 终止。

例如：

```
/*************************************************************
 * 禁止 switch 的 case 语句不是由 break 终止
 *************************************************************/
void static_p(int p_1)
{
    int i=0，j=0;
    switch(p_1)
    {
        case 0:
            j=0;
        case 1:
            j=i;
            break;
        default:
            i=j+1;
    }
    /*...*/
}
```

4.3.1.8 禁止 switch 的 case 语句中无任何可执行语句

如果某个 case 语句中无任何可执行语句，则它将共享后面 case 语句中的执行语句。这种情况或是由于代码不完整造成的，或是编程者特意设计的。为了防止残留不完整的代码，因此禁止 switch 的 case 语句中无任何可执行语句。

例如：

```
/*************************************************************
 * 禁止 switch 的 case 语句中无任何可执行语句
 *************************************************************/
int static_p(int p_1)
{
    int i=0，j=0;
    /*...*/
    switch(p_1)
    {
        case 0:
            j=0;
            break;
        case 1:
```

```
        case 2:
            j=i;
            break;
        default:
            i=j+1;
    }
    return i+j;
}
```

4.4 指针使用类

4.4.1 强制类

4.4.1.1 禁止将参数指针赋值给过程指针

将参数指针赋值给过程指针会导致不可预料的结果，因此禁止将参数指针赋值给过程指针。

例如：

```
/*****************************************************************************
 * 禁止将参数指针赋值给过程指针
 *****************************************************************************/
unsigned int *static_p(unsigned int *pl_ptr)
{
    static unsigned int w=10u;
    /*...*/
    p1_ptr=&w;
    /*...*/
    return &w;
}
```

4.4.1.2 禁止指针的指针超过两级

对指针进行控制是很困难的，当指针的指针超过两级时，使用起来更是具有风险，因此禁止指针的指针超过两级。

例如：

```
/*****************************************************************************
 * 禁止指针的指针超过两级
 *****************************************************************************/
void static_p(void)
{
    unsigned int array[10]={0};
    unsigned int *pl_ptr, **p2_ptr;
    unsigned int **p3_ptr;
    unsigned int w;
    p1_ptr=array;
    p2_ptr=&p1_ptr;
    p3_ptr=&p2_ptr;
    w=***p3_ptr;
}
```

4.4.1.3 禁止将过程声明为指针类型

使用过程指针是具有较大风险的，因此禁止将过程声明为指针类型。

例如：

```
Void foo(unsigned int p_1，unsigned short p_2)
{/*...*/}
/*****************************************
 * 禁止将过程声明为指针类型
 *****************************************/
void static_p(void)
{
    void(*proc_pointer)(unsigned int，unsigned short)=foo;
    proc_pointer(1u，1);
    /*...*/
}
```

4.4.2 推荐类

4.4.2.1 谨慎使用指针的逻辑比较

使用大于和小于的操作符对指针进行比较是具有较大风险的，应谨慎使用指针的逻辑比较。

例如：

```
/*********************************************************
 * 谨慎使用指针的逻辑比较
 *********************************************************/
void static_p(unsigned int *p1_ptr，unsigned int *p2_ptr)
{
    if (p1_ptr>p2_ptr)
    {
        /*...*/
    }
}
```

4.4.2.2 谨慎对指针进行代数运算

对指针进行代数运算是具有较大风险的，应谨慎对指针进行代数运算。

例如：

```
/*********************************************************
 * 谨慎对指针进行代数运算
 *********************************************************/
void static_p(void)
{
    unsigned int w;
    unsigned int array[5];
    unsigned int *p1_ptr;
    p1_ptr=array;
    w=*(p1_ptr+8);
}
```

4.5 跳转控制类

4.5.1 强制类

4.5.1.1 禁止直接从过程中跳出

直接从过程中跳出破坏了程序的结构化，因此禁止直接从过程中跳出。

例如：

```
#include <set jmp.h>
/*********************************************
```

```
 * 禁止直接从过程中跳出
 ********************************************/
static void static_p(jmp_buf mark，unsigned int val)
{
    longjmp(mark，val)；
}
```

4.5.1.2 禁止使用 goto 语句

使用 goto 语句是不好的编程习惯，goto 语句破坏了程序的结构化，因此禁止使用 goto 语句。

例如：

```
/*****************************************************************
 * 禁止使用 goto 语句
 *****************************************************************/
void static_p(void)
{
    int jump_flag=0；
    /*…*/
start：
    jump_flag++；
    if (jump_flag<10) {
        goto start；
    }
}
```

4.5.2 推荐类

4.5.2.1 避免使用 setjmp/longjmp

setjmp/longjmp 的使用会破坏程序的结构化，会造成移植性差，因此避免使用 setjmp/longjmp。

例如：

```
#include <setjmp.h>
/*********************************************
 * 避免使用 setjmp/longjmp
 *********************************************/
static void static_p(jmp_buf mark，unsigned int val)
{
    longjmp(mark，val)；
}
```

4.6 运算处理类

4.6.1 强制类

4.6.1.1 禁止在非赋值表达式中出现赋值操作符

在非赋值表达式中出现赋值操作符，可能是由于将“==”误写为“=”造成的，这会引起无法预料的后果，因此禁止在非赋值表达式中出现赋值操作符。

例如：

```
/*************************************************************
 * 禁止在非赋值表达式中出现赋值操作符
 *************************************************************/
void static_p(void)
{
```

```
    unsigned int z=0u，x=0u;
    bool flag=true;
    /*...*/
    if (flag=false) {
        z=x－1u;
    }
    /*...*/
}
```

4.6.1.2 数组的使用必须保证不会出现越界

该准则的违背通常是下标超出了数组所指定的范围。

例如：

```
/************************************************
 * 数组的使用必须保证不会出现越界
 ************************************************/
void static_p(void)
{
    unsigned int a[4]
    /*...*/
    a[5]=1;
    /*...*/
}
```

4.6.1.3 禁止对有符号类型进行移位运算

对有符号类型进行移位运算会导致不可预料的后果。

例如：

```
/*********************************************************************
 * 禁止对有符号类型进行移位运算
 *********************************************************************/
void static_p(void)
{
    int b=1;
    /*...*/
    b>>=1;
}
```

4.6.1.4 对变量进行移位运算必须保证不会产生溢出

一些编译器不检查移位运算是否超出机器字长。

例如：

```
/*****************************************************************
 * 对变量进行移位运算必须保证不会产生溢出
 *****************************************************************/
void static_p(void)
{
    unsigned int x=0u;
    unsigned int y=2u;
    /*...*/
    x=y<<34;
}
```

4.6.1.5 禁止给无符号变量赋负值

给无符号变量赋负值会导致不可预料的结果，因此禁止给无符号变量赋负值。

例如：

```
/*********************************************************
 * 禁止给无符号变量赋负值
 *********************************************************/
void static_p(void)
{
    unsigned int x=1u;
    unsigned int y=2u;
    /*...*/
    y=-x;
}
```

4.6.1.6 有符号类型的位长度必须大于等于两位

有符号类型只给一位的长度是没有意义的，因此有符号类型的位长度必须大于等于两位。

例如：

```
/*********************************************************************
 * 有符号类型的位长度必须大于等于两位
 *********************************************************************/
struct static_p{int x:1; };
void dummy(void)
{
    /*...*/
}
```

4.6.1.7 位的定义必须是有符号整数或无符号整数

位不能定义为有符号或无符号整数之外的其它类型。

例如：

```
/*********************************************************************
 * 位的定义必须是有符号整数或无符号整数
 *********************************************************************/
struct static_p{unsigned char x:1; };
void dummy(void)
{
    /*...*/
}
```

4.6.1.8 禁止给变量赋的值与变量的类型不一致

给变量赋的值与变量的类型不一致会导致数值有效位的损失。

例如：

```
/*********************************************************************
 * 禁止给变量赋的值与变量的类型不一致
 *********************************************************************/
void static_p(void)
{
    unsigned int d;
    d=2.0; /* Requires explicit assignment of 2u */
    /*...*/
}
```

4.6.1.9 赋值类型必须匹配

赋值类型不匹配会导致数值有效位的损失。

例如：

```
/*****************************************************************
 * 赋值类型必须匹配
 *****************************************************************/
void static_p(void)
{
    float f1=2.0f;
    double db1=3.0;
    /*...*/
    f1=db1;
}
```

4.6.1.10 数组下标必须是整型数

数组下标表示数组元素的序号，所以数组下标必须是整型数。

例如：

```
#define ArraySize 3.0f
/*****************************************************************
 * 数组下标必须是整型数
 *****************************************************************/
void static_p(void)
{
    unsigned int f1_arr[ArraySize]={0，1，2};
}
```

4.6.1.11 禁止对常数值做逻辑非的运算

对常数值做逻辑非的运算会使得逻辑判别思路混乱。

例如：

```
/*****************************************************************
 * 禁止对常数值做逻辑非的运算
 *****************************************************************/
void static_p(void)
{
    bool flag=false;
    if(flag= =!1)
    {
        /*...*/
    }
}
```

4.6.1.12 禁止对有符号类型使用位运算

位运算对有符号的数是很危险的，因为符号位会被错误地改变。

例如：

```
/*****************************************************************
 * 禁止对有符号类型使用位运算
 *****************************************************************/
void static_p(void)
{
```

```
    int b=1;
    /*...*/
    b=b | 1;
    /*...*/
}
```

4.6.1.13 禁止对枚举类型的越限使用

枚举类型只能用于与其它枚举类型进行比较，禁止对枚举类型的越限使用。

例如：

```
/*************************************************************
 * 禁止对枚举类型的越限使用
 *************************************************************/
void static_p(void)
{
    enum E_type {Enum1，Enum2，Enum3};
    unsigned int ui;
    ui=Enum1;
    /*...*/
}
```

4.6.1.14 变量的使用禁止超出所定义的范围

变量在运算过程应确保不会发生超界的数据溢出，对安全关键变量必须要仔细进行值域检查。

4.6.1.15 禁止在逻辑表达式中使用赋值操作符

在逻辑表达式中使用赋值操作符，可能是由于将“==”误写为“=”造成的，这会引起无法预料的后果，因此禁止在逻辑表达式中使用赋值操作符。

例如：

```
/*************************************************************************
 * 禁止在逻辑表达式中使用赋值操作符
 *************************************************************************/
void static_p(void)
{
    bool flag=false;
    if (flag=false)/* This condition should be(flag= =false) */
    {
    /*...*/
    }
}
```

4.6.1.16 禁止赋值操作符与“&&”或“‖”连用

这种用法是一种不好的编程习惯，因为赋值被条件化了，原期望的赋值未必能被执行，因此禁止赋值操作符与“&&”或“‖”连用。

例如：

```
/*************************************************************************
 * 禁止赋值操作符与“&&”或“‖”连用。
 *************************************************************************/
void static_p(void)
{
    bool flag=false;
```

```
    unsigned int y=0u，x=0u;
    /*…*/
    if (flag &&((x=y)= =0)) {
      /*…*/
    }
}
```

4.6.1.17 禁止位操作符带有布尔型的操作数

这种情况的出现通常是用错了操作符号，例如把“‖”误写为“｜”。

例如：

```
/*************************************************************
 * 禁止位操作符带有布尔型的操作数
 *************************************************************/
void static_p(void)
{
    unsigned int x=1u;
    bool flag=false;
    /*…*/
    if((flag ｜ (x!=0))= =false) {
        /*…*/
    }
}
```

4.6.1.18 禁止位操作符作用于布尔值

这种情况的出现通常是用错了操作符号，例如把“&&”误写为“&”。

例如：

```
/*************************************************************************
 * 禁止位操作符作用于布尔值
 *************************************************************************/
void static_p(void)
{
    unsigned int y=2u;
    bool flag=false;
    flag=flag &(y= =2u);
}
```

4.6.2 推荐类

4.6.2.1 避免使用逗号操作符

除非在参数表或循环中可以使用逗号操作符，否则逗号操作符的使用会使程序的可读性降低。

例如：

```
/*************************************************
 * 避免使用逗号操作符
 *************************************************/
void static_p(void)
{
    unsigned int x=1u;
    /*…*/
    x++，x+=1;
}
```

4.6.2.2 谨防长度操作符 sizeof 的副作用

长度操作符 sizeof 不计算操作数的值，所以对表达式做长度运算时可能会得出不可预料的结果。

例如：

```
/*****************************************************************
 * 谨防长度操作符 sizeof 的副作用
 *****************************************************************/
void static_p(void)
{
    unsigned int x=1u;
    unsigned int y=2u;
    int a=3;
    /*...*/
    a=sizeof(x=y);
}
```

4.6.2.3 谨慎使用不同类型变量的混合运算

不同类型变量的混合运算是需要类型转换的，建议编程者使用强制类型转换以明确自己的要求。

例如：

```
/*****************************************************************
 * 谨慎使用不同类型变量的混合运算
 *****************************************************************/
void static_p(void)
{
    int sx=－10;
    unsigned short usi=1;
    unsigned int ui=2u;
    float fl=2.0f;
    double db1=3.0;
    /*...*/
    ui=sx+2u;
    usi=ui+1u;
    fl=db1*fl;          /* should be fl=(float)db1*fl */
    /*...*/
}
```

4.6.2.4 避免由于设计的原因导致某些代码不能执行

该准则的违背通常表现为，做为判断的条件是一个常数值，因此使一些控制分支始终不可执行，如“if(0){...}else{...};”。

例如：

```
#define defval 0
/*****************************************************************
 * 避免由于设计的原因导致某些代码不能执行
 *****************************************************************/
void static_p(void)
{
    if (0)
    {/*...*/}
    if (defval)
```

```
    {/*...*/}
}
```

4.7 过程调用类

4.7.1 强制类

4.7.1.1 实参与形参的个数必须一致

一些编译连接器忽略实参与形参的个数检查。为了防止实参与形参匹配的错误，强制要求实参与形参的个数必须一致。

例如：

```
unsigned int test_p(unsigned int p_1， unsigned short p_2)
{
    unsigned int result=0u;
    /*...*/
    result=p_1+p_2;
    return result;
}
/*************************************************************************
 * 实参与形参的个数必须一致
 *************************************************************************/
void static_p(unsigned int p_1， unsigned short p_2)
{
    test_p(1u， 2， 3);
}
```

4.7.1.2 主过程所在文件中禁止有未被该文件中任何过程调用的子过程

主过程所在文件中如果有未被该文件中任何过程调用的子过程，那么这个子过程有可能被其它文件中的过程所调用，应该把这个子过程移到有过程调用它的那个文件中去。

例如：

```
/*************************************************************************
 * 主过程所在文件中禁止有未被该文件中任何过程调用的子过程
 *************************************************************************/
unsigned int static_p(unsigned int p)
{
    unsigned int x =1u;
    /*...*/
    x=x+p;
    return x;
}
int main(void)
{
    /*...*/
    return(0);
}
```

4.7.1.3 static 类型的过程在所在文件中必须被调用

如果一个过程被声明为 static 的类型，那么它的作用域为所在文件。若它在所在文件中没被调用，就应该把它去掉。

例如：

```
static bool static_p(unsigned int);
/*****************************************************************************
 * static 类型的过程在所在文件中必须被调用
 *****************************************************************************/
static bool static_p(unsigned int p_1)
{
    bool ret=false;
    unsigned int i=p_1+1u;
    /*...*/
    if (i= =0) {
        ret=true;
    }
    return ret;
}
void main(void)
{
    /*...*/
}
```

4.7.1.4 禁止使用被禁用的过程、函数、文件或名称

由于某些原因，有许多被禁止使用的一些特殊项，如：

a) 库中的过程或函数；

b) 库文件；

c) 特定的名称。

4.7.1.5 在不能使用 extern 的文件中禁止使用 extern

在一些文件中对 extern 的使用有严格限制，必须保证在不能使用 extern 的文件中禁止使用 extern。

例如：

```
/*****************************************************************************
 * 在不能使用 extern 的文件中禁止使用 extern
 *****************************************************************************/
/* any extern to be forbidded */
extern unsigned int undef_global;
void static_p(void)
{
    undef_global=1u;
}
```

4.7.1.6 禁止同一个表达式中调用多个相关函数

如果同一个表达式中调用多个相关函数，可能会因执行的顺序不同而产生不同的结果，因此禁止同一个表达式中调用多个相关函数。

例如：

```
unsigned int exp_1(unsigned int *p_1)
{
    unsigned int x=*p_1;
    (*p_1)=x*x;
    return(x);
}
unsigned int exp_2(unsigned int *p_1)
```

```
{
    unsigned int x=*p_1;
    (*p_1)=(x % 2);
    return(x);
}
/*****************************************************************
 * 禁止同一个表达式中调用多个相关函数
 *****************************************************************/
void static_p(void)
{
    unsigned int y=3u，x=0u;
    x=exp_1(&y)+exp_2(&y);
}
```

4.7.1.7 禁止 void 类型的过程用在表达式中使用

返回类型说明为 void 的过程用在表达式中使用是危险的，因此禁止 void 类型的过程用在表达式中使用。

例如：

```
Void foo(void)
{
    /*...*/
}
/*****************************************************************
 * 禁止 void 类型的过程用在表达式中使用
 *****************************************************************/
void static_p(void)
{
    char x;
    x=(char) foo( );
}
```

4.7.1.8 禁止 void 类型的变量作为参数进行传递

传递 void 类型的参数会导致不可预料的结果，因此禁止 void 类型的变量作为参数进行传递。

例如：

```
void void_para_func(void *p_1)
{
    /*...*/
}
/*****************************************************************
 * 禁止 void 类型的变量作为参数进行传递
 *****************************************************************/
void static_p(unsigned int p_1，unsigned short p_2)
{
    int y=0;
    void *v_ptr=&y;
    y=(int)(p_1+p_2);
    void_para_func(v_ptr);
}
```

4.7.1.9 禁止实参和形参类型不一致

为了严格的参数类型匹配检查，因此禁止实参和形参类型不一致。

例如：

```
void foo(unsigned short p_1，unsigned short p_2)
{
    /*...*/
}
/*****************************************************************************
 * 禁止实参和形参类型不一致
 *****************************************************************************/
void static_p(unsigned int p_1，unsigned short p_2)
{
    foo(p_1，p_2)；
    /*...*/
}
```

4.7.1.10 函数和原型参数类型必须一致

为了严格的函数类型匹配检查，因此函数和原型参数类型必须一致。

例如：

```
bool static_103(float up_1)；
/*****************************************************************************
 * 函数和原型参数类型必须一致
 *****************************************************************************/
bool static_p(unsigned int up_1)
{
    bool ret=false；
    /*...*/
    return ret；
}
```

4.7.2 推荐类

4.7.2.1 避免过程参数在过程调用中未被使用

过程参数在过程调用中未被使用，说明该参数没有实际的存在价值。

4.7.2.2 避免以非调用方式使用函数

在表达式中使用一个过程标识符有可能无法执行函数的调用或函数地址的分配。

例如：

```
bool test_p(void)
{
    bool retval=true；
    /*...*/
    return retval；
}
/*************************************************************
 * 避免以非调用方式使用函数
 *************************************************************/
void static_p(void)
{
    if (test_p) {
```

```
        /*...*/
    }
}
```

4.7.2.3 谨慎使用 abort，exit 等函数

这些函数会导致终止程序执行，应谨慎使用这些函数。

例如：

```
#include " process.h"
/*************************************************
 * 谨慎使用 abort，exit 等函数
 *************************************************/
void static_p(bool flag)
{
    if (flag) {
        abort();
    }
    exit(0);
}
```

4.8 语句使用类

4.8.1 强制类

4.8.1.1 禁止单独使用小写字母“l”或大写字母“O”作为变量名

小写字母“l”很容易与数字“1”混淆，大写字母“O”很容易与数字“0”混淆，因此禁止单独使用小写字母“l”或大写字母“O”作为变量名。

例如：

```
/*****************************************************************
 * 禁止单独使用小写字母“l”或大写字母“O”作为变量名
 *****************************************************************/
void static_p(void)
{
    int l=1，O=0;
    /*...*/
    l=O;
    O=1;
}
```

4.8.1.2 禁止三字母词的使用

三字母词的使用使得程序难于阅读，容易出现编程失误。

例如：

```
/*****************************************
 * 禁止三字母词的使用
 *****************************************/
void static_p(void)
??</*??<在这里代替{ */
    /*...*/
}
```

4.8.1.3 使用的八进制数必须加以注释

由于八进制数是以 0 开始的，容易与十进制的数混淆，所以使用的八进制数必须加以注释。

例如：

```
/************************************************
 * 使用的八进制数必须加以注释
 ************************************************/
void static_p(void)
{
    unsigned int i;
    i=(unsigned int)076;
    /*...*/
}
```

4.8.2 推荐类

4.8.2.1 避免使用“+=”或“-=”操作符

“+=”或“-=”操作符的使用具有简洁的特点，但也影响了可读性，容易出现编程失误。

例如：

```
/****************************************************************
 * 避免使用“+=”或“-=”操作符
 ****************************************************************/
void static_p(void)
{
    unsigned int x=1u;
    unsigned int y=2u;
    unsigned int z=3u;
    bool flag=false;
    /*...*/
    x+=1;
    z+=y;
}
```

4.8.2.2 谨慎使用“++”或“--”操作符

在表达式中使用这些操作符是很危险的。操作符在变量前的使用称之为前缀使用，操作符在变量后的使用称之为后缀使用。“++”或“--”一般只在一个单独的表达式或循环控制中使用。

例如：

```
void foo(unsigned int p_x)
{
    /*...*/
/****************************************************************
 * 谨慎使用“++”或“--”操作符
 ****************************************************************/
void static_p(void)
{
    unsigned int x=1u;
    unsigned int y=2u;
    bool flag=false;
    /*...*/
    if(flag==false)
    {
        x++;
```

```
    }
    x=x+y++;
    foo(x++);
}
```

4.8.2.3 避免使用 continue 语句

continue 语句的使用有时会导致无实际意义的程序代码，应避免使用 continue 语句。

例如：

```
/*********************************************************
 * 避免使用 continue 语句
 *********************************************************/
void static_p(int p_1)
{
    int i=p_1;
    int x=0;
    /*...*/
    while(i!=0) {
        i--;
        if(x==0) {
            continue;
        }
    }
}
```

4.8.2.4 谨慎使用三重表达式

三重表达式的使用会降低程序的可读性，容易出现编程失误。

例如：

```
void foo(unsigned int p_1)
{
    /*...*/
}
/*********************************************************************
 * 谨慎使用三重表达式
 *********************************************************************/
void static_p(unsigned int p_1)
{
    static unsigned int type0=0u;
    static unsigned int type1=1u;
    /*...*/
    (p_1==0)?foo(type0): foo(type1);
}
```

4.8.2.5 避免使用不起作用的语句

一些语句如果没有改变局部变量或全局变量的值，也没有影响控制流，说明这些语句没有实际的存在价值。

例如：

```
/*********************************************************************
 * 避免使用不起作用的语句
 *********************************************************************/
```

```
void static_p(void)
{
    unsigned int x=0u;
    x;
}
```

4.8.2.6 避免使用空语句

C 语言允许使用空语句，但有些编译器是不能处理空语句的，所以建议避免使用空语句。

例如：

```
/*************************************************
 * 避免使用空语句
 *************************************************/
void static_p(void)
{
    unsigned int Timing_Loop=100u;
    /*...*/
    Timing_Loop--;;
}
```

4.8.2.7 谨慎使用寄存器变量

寄存器变量的使用可能会产生不可预料的结果，所以应谨慎使用。

例如：

```
/*************************************************
 * 谨慎使用寄存器变量
 *************************************************/
void static_p(void)
{
    register int ri=0;
    /*...*/
}
```

4.8.2.8 避免使用老的参数表的定义形式

建议使用新的(ANSI)参数表的定义形式。

例如：

```
/*************************************************
 * 避免使用老的参数表的定义形式
 *************************************************/
void static_p(p_1, p_2)
int *p_1;
int *p_2;
{
    /*...*/
}
```

4.9 调用返回类

4.9.1 强制类

4.9.1.1 函数必须有返回语句

一个函数应该有一个返回语句，否则函数会返回一个随机数，这个随机数通常是堆栈顶端值。

例如：

```
/*****************************************************************
 * 函数必须有返回语句
 *****************************************************************/
unsigned int static_p(unsigned int p_1， unsigned short p_2)
{
    unsigned int y=p_1;
    /* Not returning a value */
}
```

4.9.1.2 禁止 void 类型的过程中的 return 语句带有返回值

如果过程返回类型为 void，则该过程的设计本身就是无返回类型值的，因此禁止 void 类型的过程中的 return 语句带有返回值。

例如：

```
/*****************************************************************
 * 禁止 void 类型的过程中的 return 语句带有返回值
 *****************************************************************/
void static_p(unsigned int p_1， unsigned int p_2)
{
    unsigned int result;
    /*...*/
    result=p_1+p_2;
    return result;
}
```

4.9.1.3 有返回值的函数中 return 必须带有返回值

有返回值的函数中如果只有 **return** 语句而无具体的指定值，则此时将返回一个随机数。

例如：

```
/*****************************************************************
 * 有返回值的函数中 return 必须带有返回值
 *****************************************************************/
unsigned int static_p(unsigned int p_1， unsigned short p_2)
{
    /*...*/
    return;
}
```

4.9.1.4 函数返回类型必须一致

既然依据设计定义了函数的返回类型，则函数的实际返回类型必须与定义的返回类型相一致。

例如：

```
/*****************************************************************
 * 函数返回类型必须一致
 *****************************************************************/
unsigned int static_p(unsigned int par_1)
{
    switch(par_1){
        case 0:
            return(－1);
            break;
        case 1:
```

```
        return(1u);
        break;
    case 2:
        return(1L);
        break;
    case 3:
        return(1.0f);
        break;
    default:
        break;
    }
}
```

4.9.1.5 函数和原型返回类型必须一致

既然依据设计声明了原型的返回类型，则函数的返回类型必须与原型的返回类型相一致。

例如：

```
float static_102(unsigned int, unsigned short);
/*****************************************************************************
 * 函数和原型返回类型必须一致
 *****************************************************************************/
int static_p(unsigned int p_1, unsigned short p_2)
{
    int result=0;
    /*...*/
    return result;
}
```

4.10 程序注释类

4.10.1 强制类

4.10.1.1 禁止使用嵌套的注释

嵌套注释是否通过依赖于具体的编译器，同时嵌套注释也影响了程序的可读性，所以禁止使用嵌套的注释。

例如：

```
/***********************************************
 * 禁止使用嵌套的注释
 ***********************************************/
void static_p(void)
{
    /* This is the Outer Comment
    /* This is the Inner Comment
    */
}
```

4.10.2 推荐类

4.10.2.1 避免使用不加以分析的注释

注释应为功能性的，而非指令的逐句原样说明，对于源文件中的代码应该给予分析性的注释。

例如：

```
/*************************************************************
 * 避免使用不加以分析的注释
 *************************************************************/
void static_p(void)
{
    unsigned int x=0u;
    /*...*/
    x=1u;
    /* Let x to be 1 */
    x=x+1u;
    /* x becomes x add 1*/
}
```

4.10.2.2 建议不使用单行注释“//”

单行注释并非在所有编译器中都是有效的，所以建议不使用单行注释“//”。

例如：

```
/*************************************************************
 * 建议不使用单行注释“//”
 *************************************************************/
void static_p(void)
{
    // This is a single line comment
    /*...*/
}
```

4.11 循环控制类

4.11.1 强制类

4.11.1.1 禁止使用不合适的循环变量类型

有许多类型不适合用于循环变量，尤其是实型变量。

例如：

```
/*************************************************************
 * 禁止使用不合适的循环变量类型
 *************************************************************/
void static_p(void)
{
    float f=0.0f;
    /*...*/
    for(f=0.0f; f<10.0f; f=f+1.0f){
        /*...*/
    }
}
```

4.11.1.2 循环变量必须是局部声明的

循环变量应该定义在最小的范围内，即循环变量的作用域应最小，所以循环变量必须是局部声明的。

例如：

```
unsigned int global_f=0u;
/*************************************************************
 * 循环变量必须是局部声明的
 *************************************************************/
```

```
int loop_standards(int p_1)
{
    int j=10;
    /*...*/
    for(global_f=0; global_f<10; global_f=global_f+1) {
        j--;
    }
    return j;
}
```

4.11.2 推荐类

4.11.2.1 谨慎使用无限循环语句

下面是几种可以引起无限循环的简单情况：

```
for(;;)
while(1){...}
do...while(1)
```

在确实需要使用无限循环语句时，要进行认真核查。

4.11.2.2 避免在循环中使用 break 语句

在循环中使用 break 语句使得循环有多于一个出口，因此避免在循环中使用 break 语句。

例如：

```
/*************************************************************
 * 避免在循环中使用 break 语句
 *************************************************************/
void static_p(void)
{
    int i=10;
    /*...*/
    while(i>-1) {
        if(i==0) {
            break;
        }
        i=i-1;
    }
}
```

4.11.2.3 谨慎使用无法控制的循环条件

用来做为循环的条件是一个常数值，可能导致无限循环，例如：while(1){...}，因此谨慎使用无法控制的循环条件。

例如：

```
#define defval 0
/*************************************************************
 * 谨慎使用无法控制的循环条件
 *************************************************************/
void static_p(void)
{
    while(0)
    {/*...*/}
```

```
    while(defval)
    {/*...*/}
}
```

4.12 类型转换类

4.12.1 强制类

4.12.1.1 禁止对指针变量使用强制类型转换赋值

强制将其它类型转换为指针类型是很危险的，因此禁止对指针变量使用强制类型转换赋值。

例如：

```
/*************************************************************
 * 禁止对指针变量使用强制类型转换赋值
 *************************************************************/
void static_p(void)
{
    unsigned short s=0;
    unsigned int *pl_ptr;
    pl_ptr=(unsigned int *)s;
    /*...*/
}
```

4.12.2 推荐类

4.12.2.1 谨慎使用其它类型变量给指针赋值

用其它类型变量给指针赋值时，要用取地址运算，否则是很危险的。

4.12.2.2 避免使用不必要的类型转换

使用不必要的类型转换说明了编程者不清晰的设计意图，将不必要的类型转换去掉也可以提高程序的执行效率。

例如：

```
/*******************************************************
 * 避免使用不必要的类型转换
 *******************************************************/
void static_p(void)
{
    int sx，sy=－10;
    sx=(int)sy+1;
    /*...*/
}
```

4.12.2.3 注意三重表达式中的类型匹配

对三重表达式中任两项的类型匹配要进行仔细检查。

4.13 初始化类

4.13.1 强制类

4.13.1.1 枚举元素的初始化必须完整

枚举类型的初始化只有两种形式是安全的。一是初始化所有的元素，二是只初始化第一个元素。

例如：

```
/*************************************************************************
 * 枚举元素的初始化必须完整
 *************************************************************************/
void static_p(void)
```

```
{
    enum E_type{numl，num2=2，num3}；
    /*...*/
}
```

4.13.1.2 结构体变量初始化的类型必须一致

结构体变量的初值类型必须与结构体变量的定义类型一致。

例如：

```
struct s_type_a {int xs；float fs；}；
/*****************************************************************************
 * 结构体变量初始化的类型必须一致
 *****************************************************************************/
void static_p(void)
{
    struct s_type_a sta={3.14f，0.0f}；
    /*...*/
}
```

4.13.1.3 结构体变量初始化的嵌套结构必须与定义的相一致

通常是编程者疏忽了结构中的结构变量的嵌套层次关系，在对结构体变量初始化时必须要保持与定义的嵌套结构相一致。

例如：

```
struct pixel {unsigned int colour；struct {unsigned int x，y；} coords；}；
/*****************************************************************************
 * 结构体变量初始化的嵌套结构必须与定义的相一致
 *****************************************************************************/
void static_p(void)
{
    unsigned int xs=0u；
    unsigned int ys=0u；
    struct pixel spot={1u，xs，ys}；
    /*...*/
}
```

4.13.1.4 变量使用前必须被赋过值

在使用变量前应确保变量已被赋值。例如：

```
unsigned int x；
unsigned int y；
y=x；
```

其中对变量 x 的使用，在使用前未被赋值。

如果变量是在某些条件前提下进行的赋值，在条件结束后使用该变量，则同样是违背该条准则。

4.14 比较判断类

4.14.1 强制类

4.14.1.1 禁止对实数类型的量做是否相等的比较

对实数类型的量做是否相等的比较是很危险的，因为实数类型的量完全相等的几率是很小的。

例如：

```
/*****************************************************************************
 * 禁止对实数类型的量做是否相等的比较
```

```
 **************************************************************************/
void static_p(void)
{
    float f1， f2;
    f1=1.01f;
    f2=2.01f;
    /*...*/
    if(f1= =f2) {
        /*...*/
    }
    if(f1= =0.0f) {
        f1=f1+0.01f;
    }
}
```

4.14.1.2 禁止逻辑判别的表达式不是逻辑表达式

逻辑判别的表达式应是逻辑表达式，逻辑表达式才真正体现了逻辑的判别。通常该准则的违背是在应使用逻辑表达式的地方使用了整数表达式。

例如：

```
/*********************************************************
 * 禁止逻辑判别的表达式不是逻辑表达式
 *********************************************************/
void static_p(void)
{
    unsigned int x=0u;
    if(x) {
        /*...*/
    }
}
```

4.14.1.3 switch 语句中的表达式禁止是逻辑表达式

因为逻辑表达式只有真和假两种情况，当需要对逻辑表达式判别时，应使用 if then else 语句，而不应是 switch 语句。

例如：

```
/*********************************************************************
 * switch 语句中的表达式禁止是逻辑表达式
 *********************************************************************/
void static_p(void)
{
    bool flag=false;
    /*...*/
    switch(flag) {
        case true:
    /*...*/
        break;
    case false:
    /*...*/
        break;
```

```
        default:
            break;
        }
}
```

4.14.2 推荐类

4.14.2.1 建议逻辑表达式采用显式的表达

从清晰可读的角度出发，建议逻辑表达式采用显式的表达。例如：

if((!(trcsta<=ISTRAC))&&(!(fdiret<=IT)))

应该写成：

if((trcsta>ISTRAC)&&(fdiret>IT))

4.15 名称、符号与变量使用类

4.15.1 强制类

4.15.1.1 禁止枚举类型中的元素名与已有的变量名同名

枚举类型中的元素名应保持唯一性。该准则的违背通常表现在，枚举中元素名与已有的全局变量名同名。

例如：

```
unsigned int duplicate=0u;
/*****************************************************************
 * 禁止枚举类型中的元素名与已有的变量名同名
 *****************************************************************/
void static_p(void)
{
    enum Name_type {e1, duplicate} EnumVar;
    EnumVar=e1;
    /*...*/
}
```

4.15.1.2 禁止局部变量与全局变量同名

C 语言编译器是允许局部变量与全局变量同名，但局部变量的作用域只限制在声明的模块内部。为避免本意是需要对全局变量更新，但由于存在同名的局部变量，导致全局变量未得到实际有效的更新，因此禁止局部变量与全局变量同名。

例如：

```
unsigned int Fire_Command;
/*****************************************************************
 * 禁止局部变量与全局变量同名
 *****************************************************************/
void static_p(void)
{
    unsigned int Fire_Command=2u;
    ......
}
```

4.15.1.3 禁止形参名与全局变量名同名

形参名与全局变量名同名使程序的可读性降低，且容易出现编程失误，因此禁止形参名与全局变量名同名。

例如：

```
/*--Global Declarations--*/
unsigned int global_int=0;
/*************************************************************************
 * 禁止形参名与全局变量名同名
 *************************************************************************/
void static_p(unsigned int *global_int)
{
    /*...*/
}
```

4.15.1.4 禁止形参名与类型或标识符同名

形参名与类型或标识符同名使程序的可读性降低，且容易出现编程失误，因此禁止形参名与类型或标识符同名。

例如：

```
typedef unsigned int DUPLICATE;
/*************************************************************************
 * 禁止形参名与类型或标识符同名
 *************************************************************************/
void static_p(unsigned int DUPLICATE)
{
    /*...*/
}
```

4.15.1.5 禁止在内部块中重定义已有的变量名

块结构允许在内部块中重定义已有的变量名，但这是不好的编程习惯，且容易出现编程失误，因此禁止在内部块中重定义已有的变量名。

例如：

```
/*********************************************************
 * 禁止在内部块中重定义已有的变量名
 *********************************************************/
void static_p(unsigned int p_1)
{
    unsigned int static_p;
    bool c_1=false;
    /*...*/
    if(c_1){
        unsigned int static_p=1u;
        /*...*/
        static_p=static_p+1u;
    } else {
        static_p=p_1;
    }
    /*...*/
}
```

4.15.1.6 禁止复杂表达式中使用 volatile 类型的变量

volatile 类型变量的值可以被随意地改变，因此只在简单的表达式中使用它才是安全的，而在复杂表达式中禁止使用 volatile 类型的变量。

例如：

```
/*****************************************************************
 * 禁止复杂表达式中使用 volatile 类型的变量
 *****************************************************************/
void static_p(void)
{
    unsigned int y=0u;  x=0u,  z=1u;
    volatile unsigned int v=1u;
    /*...*/
    x=v+z/v*y;
    /*...*/
}
```

4.15.2 推荐类

4.15.2.1 在源程序中谨慎使用非标准字符

在源文件中应谨慎使用 ANSI C 定义之外的字符。

例如：

```
/*****************************************************************
 * 在源程序中谨慎使用非标准字符
 *****************************************************************/
void static_p(void)
{
    unsigned char non_std_char='ü';
    /*...*/
}
```

4.15.2.2 在宏中谨慎使用“##”或“#”

这些操作符的使用非常技巧，很可能会因疏忽大意而导致定义不成功，所以在宏中使用“##”或“#”需要特别谨慎。

例如：

```
#include <stdio.h>
/*****************************************************
 * 在宏中谨慎使用“##”或“#”
 *****************************************************/
#define FillArray(x, y)sprintf(n[##x], " %s" , &y)
void static_p(void)
{
    char n[3][10];
    /*...*/
    FillArray(0, " TEST 0" );
    FillArray(1, " TEST 1" );
    FillArray(2, " TEST 2" );
    /*...*/
}
```

附　录　A
（资料性附录）
变量命名准则

变量命名要清晰。通常的命名有头字母大写命名法、下划线命名法和带类型说明的命名法。

a)　头字母大写命名法，如：InitialValue，ObjectPosition 等；

b)　下划线命名法，如：initial_value，object_position 等；

c)　带类型说明的头字母大写命名法，如：intInitialValue 表明是 int 的类型，flObjectPosition 表明是 float 的类型，而用 tempInitialValue 来表明其是一个临时变量。

变量的命名对程序的理解及维护起着非常重要的作用。特别是在利用文字编辑工具进行变量名替换时，如果没有一个很好的变量命名体系，则往往会出现替换错误的问题。

中 华 人 民 共 和 国
国家军用标准
航天型号软件 C 语言安全子集
GJB 5369—2005
*
国防科工委军标出版发行部出版
（北京东外京顺路 7 号）
国防科工委军标出版发行部印刷车间印刷
国防科工委军标出版发行部发行

*
开本 880×1230　1/16　印张 3½　字数 110 千字
2005 年 6 月第 1 版　2005 年 6 月第 1 次印刷
印数 1—400
*
军标出字第 5975 号　定价 28.00 元