C基础与规范

**参考资料：**

Google\_Cpp\_Style\_guide\_CN.pdf

C语言编程规范--华为技术有限公司\_7812089.pdf

高质量C++\_C编程指南（林锐博士）

MISRA-C-2004 工业标准的C编程规范（中文版）

# 一 表达式

## 1 表达式——计算次序问题

* **表达式的值在标准所允许的任何运算次序下都应该是相同的。**

说明：除了少数操作符（函数调用操作符 ( )、 &&、 | |、 ? : 和 , （逗号）） 之外，子表达式所依据的运算次序是未指定的并会随时更改。注意，运算次序的问题不能使用括号来解决，因为这不是优先级的问题。

将复合表达式分开写成若干个简单表达式，明确表达式的运算次序，就可以有效消除非预期副作用。

### 1、 自增或自减操作符

示例：

x = b[i] + i++;

b[i] 的运算是先于还是后于 i ++ 的运算，表达式会产生不同的结果，把自增运算做为单独的语句，可以避免这个问题。

x = b[i] + i;

i ++;

### 2﹑ 函数参数

说明：函数参数通常从右到左压栈，但函数参数的计算次序不一定与压栈次序相同。

示例：

x = func( i++, i);

应该修改代码明确先计算第一个参数：

i++;

x = func(i, i);

### 3、 函数指针

说明：函数参数和函数自身地址的计算次序未定义。

示例：

p->task\_start\_fn(p++);

求函数地址p与计算p++无关，结果是任意值。必须单独计算p++：

p->task\_start\_fn(p);

p++;

### 4﹑ 函数调用

示例：

int g\_var = 0;

int fun1()

{

g\_var += 10;

return g\_var;

}

int fun2()

{

g\_var += 100;

return g\_var;

}

int x = fun1() + fun2();

编译器可能先计算fun1()，也可能先计算fun2()， 由于x的结果依赖于函数fun1()/fun2()的计算次序（fun1()/fun2()被调用时修改和使用了同一个全局变量），则上面的代码存在问题。

应该修改代码明确fun1/ fun2的计算次序：

int x = fun1();

x = x + fun2();

### 5、 嵌套赋值语句

说明：表达式中嵌套的赋值可以产生附加的副作用。不给这种能导致对运算次序的依赖提供任何机会的最好做法是，不要在表达式中嵌套赋值语句。

示例：

~~x = y = y = z / 3;~~

~~x = y = y++;~~

### 6、 volatile访问

说明：限定符volatile表示可能被其它途径更改的变量，例如硬件自动更新的寄存器。编译器不会优化对volatile变量的读取。

示例：**下面的写法可能无法实现作者预期的功能**：

/\* volume变量被定义为volatile类型\*/

UINT16 x = ( volume << 3 ) | volume; /\* 在计算了其中一个子表达式的时候， volume

的值可能已经被其它程序或硬件改变，导致另外一个子表达式的计算结果非预期，可能

无法实现作者预期的功能\*/

## 2 表达式——函数间调用

* **函数调用不要作为另一个函数的参数使用，否则对于代码的调试、阅读都不利。**

说明： 如下代码不合理， 仅用于说明当函数作为参数时，由于参数压栈次数不是代码可以控制的，可能造成未知的输出：

int g\_var;

int fun1()

{

g\_var += 10;

return g\_var;

}

int fun2()

{

g\_var += 100;

return g\_var;

}

int main(int argc, char \*argv[], char \*envp[])

{

g\_var = 1;

printf("func1: %d, func2: %d\n", ~~fun1(), fun2())~~;

g\_var = 1;

printf("func2: %d, func1: %d\n", ~~fun2(), fun1()~~);

}

上面的代码，使用断点调试起来也比较麻烦，阅读起来也不舒服，所以不要为了节约代码行，而写这种代码。

## 3 表达式——赋值语句

* **赋值语句不要写在if等语句中，或者作为函数的参数使用。**

说明： 因为if语句中，会根据条件依次判断，如果前一个条件已经可以判定整个条件，则后续条件语句不会再运行，所以可能导致期望的部分赋值没有得到运行。

示例：

int main(int argc, char \*argv[], char \*envp[])

{

int a = 0;

int b;

if ((a == 0) || ((b = fun1()) > 10))

{

printf("a: %d\n", a);

}

printf("b: %d\n", b);

}

作用函数参数来使用，参数的压栈顺序不同可能导致结果未知。

看如下代码，能否一眼看出输出结果会是什么吗？好理解吗？

int g\_var;

int main(int argc, char \*argv[], char \*envp[])

{

g\_var = 1;

printf("set 1st: %d, add 2nd: %d\n", g\_var = 10, g\_var++);

g\_var = 1;

printf("add 1st: %d, set 2nd: %d\n", g\_var++, g\_var = 10);

}

## 4 表达式——括号与默认优先级

* **用括号明确表达式的操作顺序，避免过分依赖默认优先级。**

说明：使用括号强调所使用的操作符，防止因默认的优先级与设计思想不符而导致程序出错；同时使得代码更为清晰可读，然而过多的括号会分散代码使其降低了可读性。下面是**如何使用括号的建议**。

**1. 一元操作符，不需要使用括号**

x = ~a; /\* 一元操作符，不需要括号\*/

x = -a; /\* 一元操作符，不需要括号\*/

**2. 二元以上操作符，如果涉及多种操作符，则应该使用括号**

x = a + b + c; /\* 操作符相同，不需要括号\*/

x = f ( a + b, c ) /\* 操作符相同，不需要括号\*/

if (a && b && c) /\* 操作符相同，不需要括号\*/

x = (a \* 3) + c + d; /\* 操作符不同，需要括号\*/

x = ( a == b ) ? a : ( a –b ); /\* 操作符不同，需要括号\*/

**3 .即使所有操作符都是相同的，如果涉及类型转换或者量级提升，也应该使用括号控制计算的次序以下代码将3个浮点数相加：**

/\* 除了逗号(,)，逻辑与(&&)，逻辑或(||)之外， C标准没有规定同级操作符是从左还是 从右开始计算，以上表达式存在种计算次序： f4 = (f1 + f2) + f3 或f4 = f1 + (f2 + f3)，

浮点数计算过程中可能四舍五入，量级提升，计算次序的不同会导致f4的结果不同， 以上表达式在不同编译器上的计算结果可能不一样，建议增加括号明确计算顺序\*/

f4 = f1 + f2 + f3;

.

## 5 表达式——赋值操作与布尔值

* **赋值操作符不能使用在产生布尔值的表达式上。**

说明：如果布尔值表达式需要赋值操作，那么赋值操作必须在操作数之外分别进行。 这可以帮助避免=和= =的混淆，帮助我们静态地检查错误。

示例：

/\* 正确示范 \*/

x = y;

if (x != 0)

{

foo ();

}

不能写成：

if (( x = y ) != 0)

{

foo ();

}

或者更坏的

if (x = y)

{

foo ();

}