[概述]

C 语言像一把雕刻刀，锋利，并且在技师手中非常有用。和任何锋利工具同样，C 会伤到那些不能掌握它人。本文简介 C 语言伤害粗心人办法，以及如何避免伤害。

[内容]

参照

脚注

0 简介

C 语言及其典型实现被设计为能被专家们容易地使用。这门语言简洁并附有表达力。但有某些限制可以保护那些浮躁人。一种浮躁人可以从这些条款中获得某些协助。

在本文中，咱们将会看到这些未可知益处。正是由于它未可知，咱们无法为其进行完全分类。但是，咱们依然通过研究为了一种 C 程序运营所需要做事来做到这些。咱们假设读者对 C 语言至少有个粗浅理解。

第一某些研究了当程序被划分为记号时会发生问题。第二某些继续研究了当程序记号被编译器组合为声明、表达式和语句时会浮现问题。第三某些研究了由各种某些构成、分别编译并绑定到一起 C 程序。第四某些解决了概念上误解：当一种程序详细执行时会发生事情。第五某些研究了咱们程序和它们所使用惯用库之间关系。在第六某些中，咱们注意到了咱们所写程序也许并不是咱们所运营程序；预解决器将一方面运营。最后，第七某些讨论了可移植性问题：一种能在一种实现中运营程序无法在另一种实现中运营因素。

1 词法缺陷

编译器第一种某些常被称为词法分析器（lexical analyzer）。词法分析器检查构成程序字符序列，并将它们划分为记号（token）一种记号是一种由一种或各种字符构成序列，它在语言被编译时具备一种（有关地）统一意义。在 C 中， 例如，记号->意义和构成它每个独立字符具备明显区别，并且其意义独立于->浮现上下文环境。

此外一种例子，考虑下面语句：

```
if(x > big) big = x;
```

该语句中每一种分离字符都被划分为一种记号，除了核心字 if 和标记符 big 两个实例。

事实上，C 程序被两次划分为记号。一方面是预解决器读取程序。它必要对程序进行记号划分以发现标记宏标记符。它必要通过对每个宏进行求值来替代宏调用。最后，通过宏替代程序又被汇集成字符流送给编译器。编译器再第二次将这个流划分为记号。

在这一节中，咱们将摸索对记号意义普遍误解以及记号和构成它们字符之间关系。稍后咱们将谈到预解决器。

1.1 = 不是 ==

从 Algol 派生出来语言，如 Pascal 和 Ada，用:=表达赋值而用=表达比较。而 C 语言则是用=表达赋值而用==表达比较。这是由于赋值频率要高于比较，因而为其分派更短符号。

此外，C 还将赋值视为一种运算符，因而可以很容易地写出多重赋值（如 a = b = c），并且可以将赋值嵌入到一种大表达式中。

这种便捷导致了一种潜在问题：也许将需要比较地方写成赋值。因而，下面语句好像看起来是要检查 x 与否等于 y：

```
if(x = y)
    foo();
```

而事实上是将 x 设立为 y 值并检查成果与否非零。再考虑下面一种但愿跳过空格、制表符和换行符循环：

```
while(c == ' ' || c = '\t' || c == '\n')
    c = getc(f);
```

在与'\t'进行比较地方程序员错误地使用=代替了==。这个“比较”事实上是将'\t'赋给 c，然后判断 c（新）值与否为零。由于'\t'不为零，这个“比较”将始终为真，因而这个循环会吃尽整个文献。这之后会发生什么取决于特定实现与否容许一种程序读取超过文献尾部某些。如果容许，这个循环会始终运营。

某些 C 编译器会对形如 e1 = e2 条件给出一种警告以提示顾客。当你的确需要先对一种变量进行赋值之后再检查变量与否非零时，为了在这种编译器中避免警告信息，应考虑显式给出比较符。换句话说，将:

```
if(x = y)
    foo();
```

改写为：

```
if((x = y) != 0)
    foo();
```

这样可以清晰地表达你意图。

1.2 & 和 | 不是 && 和 ||

容易将==错写为=是由于诸多其她语言使用=表达比较运算。 其她容易写错运算符尚有&和&&，以及|和||，这重要是由于 C 语言中&和|运算符于其她语言中具备类似功能运算符大为不同。咱们将在第 4 节中贴近地观测这些运算符。

1.3 多字符记号

某些 C 记号，如/、*和=只有一种字符。而其她某些 C 记号，如/*和==，以及标记符，具备各种字符。当 C 编译器遇到紧连在一起/和*时，它必要可以决定是将这两个字符辨以为两个分离记号还是一种单独记号。C 语言参照手册阐明了如何决定：“如果输入流到一种给定字符串为止已经被辨以为记号，则应当包括下一种字符以构成可以构成记号最长字符串”（[译注]即普通所说“最长子串原则”）。因而，如果/是一种记号第一种字符，并且/背面紧随了一种*，则这两个字符构成了注释开始，不论其她上下文环境。

下面语句看起来像是将 y 值设立为 x 值除以 p 所指向值：

```
y = x/*p      /* p 指向除数 */;
```

事实上，/*开始了一种注释，因而编译器简朴地吞噬程序文本，直到*/浮现。换句话说，这条语句仅仅把 y 值设立为 x 值，而主线没有看到 p。将这条语句重写为：

```
y = x / *p      /* p  指向除数  */;
```

或者干脆是

```
y = x / (*p)      /* p 指向除数  */;
```

它就可以做注释所暗示除法了。

这种模棱两可写法在其她环境中就会引起麻烦。例如，老版本 C 使用=+表达当前版本中+=。这样编译器会将

```
a=-1;
```

视为

```
a =- 1;
```

或

```
a = a - 1;
```

这会让打算写

```
a = -1;
```

程序员感到吃惊。

另一方面，这种老版本 C 编译器会将

a=/*b;

断句为

a =/ *b;

尽管/*看起来像一种注释。

1.4 例外

组合赋值运算符如+=事实上是两个记号。因而，

a + /* strange */ = 1

和

a += 1

是一种意思。看起来像一种单独记号而事实上是各种记号只有这一种特例。特别地，

p - > a

是不合法。它和

p -> a

不是同义词。

另一方面，有些老式编译器还是将=+视为一种单独记号并且和+=是同义词。

1.5 字符串和字符

单引号和双引号在 C 中意义完全不同，在某些混乱上下文中它们会导致奇怪成果而不是错误消息。

包围在单引号中一种字符只是编写整数另一种办法。这个整数是给定字符在实现对照序列中一种相应值。因而，在一种 ASCII 实现中，'a'和 0141 或 97 表达完全相似东西。而一种包围在双引号中字符串，只是编写一种有双引号之间字符和一种附加二进制值为零字符所初始化一种无名数组指针一种简短办法。

下面两个程序片断是等价：

```
printf("Hello world\n");
```

```
char hello[] = { 'H'，'e'，'l'，'l'，'o'，' '，'w'，'o'，'r'，'l'，'d'，'\n'，0 };
printf(hello);
```

使用一种指针来代替一种整数普通会得到一种警告消息（反之亦然），使用双引号来代替单引号也会得到一种警告消息（反之亦然）。但对于不检查参数类型编译器却除外。因而，用

```
printf('\n');
```

来代替

```
printf("\n");
```

普通会在运营时得到奇怪成果。（[译注]提示：正如上面所说，'\n'表达一种整数，它被转换为了一种指针，这个指针所指向内容是没故意义。）

由于一种整数普通足够大，以至于可以放下各种字符，某些 C 编译器容许在一种字符常量中存储各种字符。这意味着用'yes'代替"yes"将不会被发现。后者意味着“分别包括 y、e、s 和一种空字符四个持续存储器区域中第一种地址”，而前者意味着“在某些实现定义样式中表达由字符 y、e、s 联合构成一种整数”。这两者之间任何一致性都纯属巧合。

2 句法缺陷

要理解 C 语言程序，仅理解构成它记号是不够。还要理解这些记号是如何构成声明、表达式、语句和程序。尽管这些构成普通都是定义良好，但这些定义有时候是有悖于直觉或混乱。

在这一节中，咱们将着眼于某些不明显句法构造。

2.1 理解声明

我曾经和某些人聊过天，她们那时正在在编写在一种小型微解决器上单机运营 C 程序。当这台机器开关打开时候，硬件会调用地址为 0 处子程序。

为了模仿电源打开情形，咱们要设计一条 C 语句来显式地调用这个子程序。通过某些思考，咱们写出了下面语句：

```
(*(void(*)())0)();
```

这样表达式会令 C 程序员心惊胆战。但是，并不需要这样，由于她们可以在一种简朴规则协助下很容易地构造它：以你使用方式声明它。

每个 C 变量声明都具备两个某些：一种类型和一组具备特定格式、盼望用来对该类型求值表达式。最简朴表达式就是一种变量：

```
float f，g;
```

阐明表达式 f 和 g——在求值时候——具备类型 float。由于待求值是表达式，因而可以自由地使用圆括号：

```
float ((f));
```

这表达((f))求值为 float 并且因而，通过推断，f 也是一种 float。

同样逻辑用在函数和指针类型。例如：

```
float ff();
```

表达表达式 ff()是一种 float，因而 ff 是一种返回一种 float 函数。类似地，

float *pf;

表达*pf 是一种 float 并且因而 pf 是一种指向一种 float 指针。

这些形式组合声明对表达式是同样。因而，

float *g()，(*h)();

表达*g()和(*h)()都是 float 表达式。由于()比*绑定得更紧密，*g()和*(g())表达同样东西：g 是一种返回指 float 指针函数，而 h 是一种指向返回 float 函数指针。

当咱们懂得如何声明一种给定类型变量后来，就可以很容易地写出一种类型模型（cast）：只要删除变量名和分号并将所有东西包围在一对圆括号中即可。因而，由于

float *g();

声明 g 是一种返回 float 指针函数，因此(float *())就是它模型。

有了这些知识武装，咱们当前可以准备解决(*(void(*)())0)()了。 咱们可以将它分为两个某些进行分析。一方面，假设咱们有一种变量 fp，它包括了一种函数指针，并且咱们但愿调用 fp 所指向函数。可以这样写：

(*fp)();

如果 fp 是一种指向函数指针，则*fp 就是函数自身，因而(*fp)()是调用它一种办法。(*fp)中括号是必要，否则这个表达式将会被分析为*(fp())。咱们当前要找一种恰当表达式来替代 fp。

这个问题就是咱们第二步分析。如果 C 可以读入并理解类型，咱们可以写：

```
(*0)();
```

但这样并不行，由于*运算符规定必要有一种指针作为它操作数。此外，这个操作数必要是一种指向函数指针，以保证*成果可以被调用。因而，咱们需要将 0 转换为一种可以描述“指向一种返回 void 函数指针”类型。

如果 fp 是一种指向返回 void 函数指针，则(*fp)()是一种 void 值，并且它声明将会是这样：

```
void (*fp)();
```

因而，咱们需要写：

```
void (*fp)();
(*fp)();
```

来声明一种哑变量。一旦咱们懂得了如何声明该变量，咱们也就懂得了如何将一种常数转换为该类型：只要从变量声明中去掉名字即可。因而，咱们像下面这样将 0 转换为一种“指向返回 void 函数指针”：

```
(void(*)())0
```

接下来，咱们用(void(*)())0 来替代 fp：

```
(*(void(*)())0)();
```

结尾处分号用于将这个表达式转换为一种语句。

在这里，咱们解决这个问题时没有使用 typedef 声明。通过使用它，咱们可以更清晰地解决这个问题：

```
typedef void (*funcptr)();
(*(funcptr)0)();
```

2.2 运算符并不总是具备你所想象优先级

假设有一种声明了常量 FLAG，它是一种整数，其二进制表达中某一位被置位（换句话说，它是 2 某次幂），并且你但愿测试一种整型变量 flags 该位与否被置位。普通写法是：

```
if(flags & FLAG) ...
```

其意义对于诸多 C 程序员都是很明确：if 语句测试括号中表达式求值成果与否为 0。出于清晰目咱们可以将它写得更明确：

```
if(flags & FLAG != 0) ...
```

这个语句当前更容易理解了。但它依然是错，由于!=比&绑定得更紧密，因而它被分析为：

```
if(flags & (FLAG != 0)) ...
```

这（偶尔）是可以，如 FLAG 是 1 或 0（！）时候，但对于其她 2 幂是不行[2]。

假设你有两个整型变量，h 和 l，它们值在 0 和 15（含 0 和 15）之间，并且你但愿将 r 设立为 8 位值，其低位为 l，高位为 h。一种自然写法是：

```
r = h << 4 + 1;
```

不幸是，这是错误。加法比移位绑定得更紧密，因而这个例子等价于：

```
r = h << (4 + l);
```

对的办法有两种：

```
r = (h << 4) + l;
```

r = h << 4 | l;

避免这种问题一种办法是将所有东西都用括号括起来，但表达式中括号过度就会难以理解，因而最佳还是是记住 C 中优先级。

不幸是，这有 15 个，太困难了。然而，通过将它们分组可以变得容易。

绑定得最紧密运算符并不是真正运算符：下标、函数调用和构造选取。这些都与左边有关联。

接下来是一元运算符。它们具备真正运算符中最高优先级。由于函数调用比一元运算符绑定得更紧密，你必要写(*p)()来调用 p 指向函数：*p()表达 p 是一种返回一种指针函数。转换是一元运算符，并且和其她一元运算符具备相似优先级。一元运算符是右结合，因而*p++表达*(p++)，而不是(*p)++。

在接下来是真正二元运算符。其中数学运算符具备最高优先级，然后是移位运算符、关系运算符、逻辑运算符、赋值运算符，最后是条件运算符。需要记住两个重要东西是：

所有逻辑运算符具备比所关于系运算符都低优先级。

移位运算符比关系运算符绑定得更紧密，但又不如数学运算符。

在这些运算符类别中，有某些奇怪地方。乘法、除法和求余具备相似优先级，加法和减法具备相似优先级，以及移位运算符具备相似优先级。

尚有就是六个关系运算符并不具备相似优先级：==和!=优先级比其她关系运算符要低。这就容许咱们判断 a 和 b 与否具备与 c 和 d 相似顺序，例如：

a < b == c < d

在逻辑运算符中，没有任何两个具备相似优先级。按位运算符比所有顺序运算符绑定得都紧密，每种与运算符都比相应或运算符绑定得更紧密，并且按位异或（^）运算符介于按位与和按位或之间。

三元运算符优先级比咱们提到过所有运算符优先级都低。这可以保证选取表达式中包括关系运算符逻辑组合特性，如：

```
z = a < b && b < c ?d ： e
```

这个例子还阐明了赋值运算符具备比条件运算符更低优先级是故意义。此外，所有复合赋值运算符具备相似优先级并且是自右至左结合，因而

```
a = b = c
```

和

```
b = c；a = b;
```

是等价。

具备最低优先级是逗号运算符。这很容易理解，由于逗号普通在需要表达式而不是语句时候用来代替分号。

赋值是另一种运算符，普通具备混合优先级。例如，考虑下面这个用于复制文献循环：

```
while(c = getc(in) != EOF)
    putc(c，out);
```

这个 while 循环中表达式看起来像是 c 被赋以 getc(in)值，接下来判断与否等于 EOF 以结束循环。不幸是，赋值优先级比任何比较操作都低，因而 c 值将会是 getc(in)和 EOF 比较成果，并且会被抛弃。因而，“复制”得到文献将是一种由值为 1 字节流构成文献。

上面这个例子对的写法并不难：

```
while((c = getc(in)) != EOF)
```

```
    putc(c，out);
```

然而，这种错误在诸多复杂表达式中却很难被发现。例如，随 UNIX 系统一同发布 lint 程序普通带有下面错误行：

```
if (((t = BTYPE(pt1->aty) == STRTY) || t == UNIONTY) {
```

这条语句但愿给 t 赋一种值，然后看 t 与否与 STRTY 或 UNIONTY 相等。而实际效果却大不相似[3]。

C 中逻辑运算符优先级具备历史因素。B 语言——C 前辈——具备和 C 中&和|运算符相应逻辑运算符。尽管它们定义是按位 ，但编译器在条件判断上下文中将它们视为和&&和||同样。当在 C 中将它们分开后，优先级变化是很危险[4]。

2.3 看看这些分号！

C 中一种多余分号普通会带来一点点不同：或者是一种空语句，无任何效果；或者编译器也许提出一种诊断消息，可以以便除去掉它。一种重要区别是在必要跟有一种语句 if 和 while 语句中。考虑下面例子：

```
if(x[i] > big);
    big = x[i];
```

这不会发生编译错误，但这段程序意义与：

```
if(x[i] > big)
    big = x[i];
```

就大不相似了。第一种程序段等价于：

```
if(x[i] > big) { }
big = x[i];
```

也就是等价于：

```
big = x[i];
```

（除非 x、i 或 big 是带有副作用宏）。

另一种因分号引起巨大不同地方是函数定义前面构造声明末尾（[译注]这句话不太好听，看例子就明白了）。考虑下面程序片段：

```
struct foo {
    int x;
}

f() {
    ...
}
```

在紧挨着 f 第一种}背面丢失了一种分号。它效果是声明了一种函数 f，返回值类型是 struct foo，这个构导致了函数声明一某些。如果这里浮现了分号，则 f 将被定义为具备默认整型返回值[5]。

2.4 switch 语句

普通 C 中 switch 语句中 case 段可以进入下一种。例如，考虑下面 C 和 Pascal 程序片断：

```
switch(color) {
case 1：printf ("red");
        break;
case 2：printf ("yellow");
        break;
case 3：printf ("blue");
```

```
        break;
}

case color of
1：write ('red');
2：write ('yellow');
3：write ('blue');
end
```

这两个程序片段都作相似事情：依照变量 color 值是 1、2 还是 3 打印 red、yellow 或 blue（没有新行符）。这两个程序片段非常相似，只有一点不同：Pascal 程序中没有 C 中相应 break 语句。C 中 case 标签是真正标签：控制流程可以无限制地进入到一种 case 标签中。

看看另一种形式，假设 C 程序段看起来更像 Pascal：

```
switch(color) {
case 1：printf ("red");
case 2：printf ("yellow");
case 3：printf ("blue");
}
```

并且假设 color 值是 2。则该程序将打印 yellowblue，由于控制自然地转入到下一种 printf()调用。

这既是 C 语言 switch 语句长处又是它弱点。说它是弱点，是由于很容易忘掉一种 break 语句，从而导致程序浮现隐晦异常行为。说它是长处，是由于通过故意去掉 break 语句，可以很容易实现其她办法难以实现控制构造。特别是在一种大型 switch 语句中，咱们经常发现对一种 case 解决可以简化其她某些特殊解决。

例如，设想有一种程序是一台假想机器翻译器。这样一种程序也许包括一种 switch 语句来解决各种操作码。在这样一台机器上，普通减法在对其第二个运算数进行变号后就变成和加法同样了。因而，最佳可以写出这样语句：

```
case SUBTRACT:
    opnd2 = -opnd2;
    /* no break：*/
case ADD:
    ...
```

此外一种例子，考虑编译器通过跳过空白字符来查找一种记号。这里，咱们将空格、制表符和新行符视为是相似，除了新行符还要引起行计数器增长外：

```
case '\n':
    linecount++;
    /* no break */
case '\t':
case ' ':
    ...
```

2.5 函数调用

和其她程序设计语言不同，C 规定一种函数调用必要有一种参数列表，但可以没有参数。因而，如果 f 是一种函数，

```
f();
```

就是对该函数进行调用语句，而

```
f;
```

什么也不做。它会作为函数地址被求值，但不会调用它[6]。

2.6 悬挂 else 问题

在讨论任何语法缺陷时咱们都不会忘掉提到这个问题。尽管这一问题不是 C 语言所独有，但它依然伤害着那些有着近年经验 C 程序员。

考虑下面程序片断：

```
if(x == 0)
    if(y == 0) error();
else {
    z = x + y;
    f(&z);
}
```

写这段程序程序员目明显是将状况分为两种：x = 0 和 x != 0。在第一种状况中，程序段什么都不做，除非 y = 0 时调用 error()。第二种状况中，程序设立 z = x + y 并以 z 地址作为参数调用 f()。

然而， 这段程序实际效果却大为不同。其因素是一种 else 总是与其近来 if 有关联。如果咱们但愿这段程序可以按照实际状况运营，应当这样写：

```
if(x == 0) {
    if(y == 0)
        error();
    else {
        z = x + y;
```

```
        f(&z);
    }
}
```

换句话说，当 x != 0 发生时什么也不做。如果要达到第一种例子效果，应当写：

```
if(x == 0) {
    if(y ==0)
        error();
}
else {
    z = z + y;
    f(&z);
}
```

3 连接

一种 C 程序也许有诸多某些构成，它们被分别编译，并由一种普通称为连接器、连接编辑器或加载器程序绑定到一起。由于编译器一次普通只能看到一种文献，因而它无法检测到需要程序各种源文献内容才干发现错误。

在这一节中，咱们将看到某些这种类型错误。有某些 C 实现，但不是所有，带有一种称为 lint 程序来捕获这些错误。如果具备一种这样程序，那么无论如何地强调它重要性都但是分。

3.1 你必要自己检查外部类型

假设你有一种 C 程序，被划分为两个文献。其中一种包括如下声明：

```
int n;
```

而令一种包括如下声明：

long n;

这不是一种有效 C 程序，由于某些外部名称在两个文献中被声明为不同类型。然而，诸多实现检测不到这个错误，由于编译器在编译其中一种文献时并不懂得另一种文献内容。因而，检查类型工作只能由连接器（或某些工具程序如 lint）来完毕；如果操作系统连接器不能辨认数据类型，C 编译器也没法过多地强制它。

那么，这个程序运营时实际会发生什么？这有诸多也许性：

实现足够聪颖，可以检测到类型冲突。则咱们会得到一种诊断消息，阐明 n 在两个文献中具备不同类型。

你所使用实现将 int 和 long 视为相似类型。典型状况是机器可以自然地进行 32 位运算。在这种状况下你程序或允许以工作，好象你两次都将变量声明为 long（或 int）。但这种程序工作纯属偶尔。

n 两个实例需要不同存储，它们以某种方式共享存储区，即对其中一种赋值对另一种也有效。这也许发生，例如，编译器可以将 int 安排在 long 低位。无论这是基于系统还是基于机器，这种程序运营同样是偶尔。

n 两个实例以另一种方式共享存储区，即对其中一种赋值效果是对另一种赋以不同值。在这种状况下，程序也许失败。

这种状况发生里一种例子出奇地频繁。程序某一种文献包括下面声明：

char filename[] = "etc/passwd";

而另一种文献包括这样声明：

char *filename;

尽管在某些环境中数组和指针行为非常相似，但它们是不同。在第一种声明中，filename 是一种字符数组名字。尽管使用数组名字可以产生数组第一种元素指针，但这个指针只有在需要时候才产生并且不会持续。在第二个声明中，filename 是一种指针名字。这个指针可以指向程序员让它指向任何地方。如果程序员没有给它赋一种值，它将具备一种默认 0 值（NULL）（[译注]事实上，在 C 中一种为初始化指针普通具备一种随机值，这是很危险！）。

这两个声明以不同方式使用存储区，它们不也许共存。

避免这种类型冲突一种办法是使用像 lint 这样工具（如果可以话）。为了在一种程序不同编译单元之间检查类型冲突，某些程序需要一次看到其所有某些。典型编译器无法完毕，但 lint 可以。

避免该问题另一种办法是将外部声明放到包括文献中。这时，一种外部对象类型仅浮现一次[7]。

4 语义缺陷

一种句子可以是精准拼写并且没有语法错误，但依然没故意义。在这一节中，咱们将会看到某些程序写法会使得它们看起来是一种意思，但事实上是另一种完全不批准思。

咱们还要讨论某些表面上看起来合理但事实上会产生未定义成果环境。咱们这里讨论东西并不保证可以在所有 C 实现中工作。咱们暂且忘掉这些可以在某些实现中工作但也许不能在另某些实现中工作东西，直到第 7 节讨论可以执行问题为止。

4.1 表达式求值顺序

某些 C 运算符以一种已知、特定顺序对其操作数进行求值。但另某些不能。例如，考虑下面表达式：

```
a < b && c < d
```

C 语言定义规定 a < b 一方面被求值。如果 a 的确不大于 b，c < d 必要紧接着被求值以计算整个表达式值。但如果 a 不不大于或等于 b，则 c < d 主线不会被求值。

要对 a < b 求值，编译器对 a 和 b 求值就会有一种先后。但在某些机器上，它们也许是并行进行。

C 中只有四个运算符&&、||、?:和,指定了求值顺序。&&和||最先对左边操作数进行求值，而右边操作数只有在需要时候才进行求值。而?:运算符中三个操作数：a、b 和 c，最先对 a 进行求值，之后仅对 b 或 c 中一种进行求值，这取决于 a 值。,运算符一方面对左边操作数进行求值，然后抛弃它值，对右边操作数进行求值[8]。

C 中所有其他运算符对操作数求值顺序都是未定义。事实上，赋值运算符不对求值顺序做出任何保证。

出于这个因素，下面这种将数组 x 中前 n 个元素复制到数组 y 中办法是不可行：

```
i = 0;
while(i < n)
    y[i] = x[i++];
```

其中问题是 y[i]地址并不保证在 i 增长之前被求值。在某些实现中，这是也许; 但在另某些实现中却不也许。另一种状况出于同样因素会失败：

```
i = 0;
while(i < n)
    y[i++] = x[i];
```

而下面代码是可以工作：

```
i = 0;
while(i < n) {
    y[i] = x[i];
    i++;
}
```

固然，这可以简写为：

```
for(i = 0；i < n；i++)
    y[i] = x[i];
```