

- [头条](#)
- [博客](#)
- [资源](#)
- [翻译](#)
- [小组](#)
- [相亲](#)
- [注册](#)
- [登录](#)



- [首页](#)
- [最新文章](#)
- [在线课程](#)
- [业界](#)
- [开发](#)
- [IT技术](#)
- [设计](#)
- [创业](#)
- [IT职场](#)
- [在国外](#)
- [频道](#)
- [更多 >](#)

- 导航条 - ▼

[伯乐在线](#) > [首页](#) > [所有文章](#) > [C/C++](#) > C语言未定义行为一览

C语言未定义行为一览

2013/05/08 | 分类: [C/C++](#), [开发](#) | [15 条评论](#) | 标签: [C语言](#)

分享到:

67

[Java中的文件上传下载](#)
[CSS workflow](#)

[高性能产品的必由之路—性能测试工具](#)
[谈谈CSS性能](#)

本文由 [伯乐在线](#) - [cipan](#) 翻译。未经许可，禁止转载！
英文出处: [Christopher Cole](#)。欢迎加入[翻译小组](#)。

几周前，我的一位同事带着一个编程问题来到我桌前。最近我们一直在互相考问C语言的知识，所以我微笑着鼓起勇气面对无疑即将到来的地狱。

他在白板上写了几行代码，并问这个程序会输出什么？

```
1  #include <stdio.h>
2
3  int main(){
4      int i = 0;
5      int a[] = {10,20,30};
6
7      int r = 1 * a[i++] + 2 * a[i++] + 3 * a[i++];
8      printf("%d\n", r);
9      return 0;
10 }
```

看上去相当简单明了。我解释了操作符的优先顺序——后缀操作比乘法先计算、乘法比加法先计算，并且乘法和加法的结合性都是从左到右，于是我抓出运算符号并开始写出算式。

```
1  int r = 1 * a[i++] + 2 * a[i++] + 3 * a[i++];
2  //      =      a[0]      + 2 * a[1]  + 3 * a[2];
3  //      =      10       +      40    +      90;
4  //      = 140
```

我自鸣得意地写下答案后，我的同事回应了一个简单的“不”。我想了几分钟后，还是被难住了。我不太记得后缀操作符的结合顺序了。此外，我知道那个顺序甚至不会改变这里的值计算的顺序，因为结合规则只会应用于同级的操作符之间。但我想到了应该根据后缀操作符都从右到左求值的规则，尝试算一遍这条算式。看上去相当简单明了。

```
1  int r = 1 * a[i++] + 2 * a[i++] + 3 * a[i++];
2  //      =      a[2]      + 2 * a[1]  + 3 * a[0];
```

```

3 | // = 30 + 40 + 30;
4 | // = 100

```

我的同事再一次回答说，答案仍是错的。这时候我只好认输了，问他答案是什么。这段短小的样例代码原来是从他写过的更大的代码段里删减出来的。为了验证他的问题，他编译并且运行了那个更大的代码样例，但是惊奇地发现那段代码没有按照他预想的运行。他删减了不需要的步骤后得到了上面的样例代码，用gcc 4.7.3编译了这段样例代码，结果输出了令人吃惊的结果：“60”。

这时我被迷住了。我记得，C语言里，函数参数的计算求值顺序是未定义的，所以我们以为后缀操作符只是遵照某个随机的、而非从左至右的顺序，计算的。我们仍然确信后缀比加法和乘法拥有更高的操作优先级，所以很快证明我们自己，不存在我们可以计算i++的顺序，使得这三个数组元素一起加起来、乘起来得到60。

现在我已对此入迷了。我的第一个想法是，查看这段代码的反汇编代码，然后尝试查出它实际上发生了什么。我用调试符号（debugging symbols）编译了这段样例代码，用了objdump后很快得到了带注释的x86_64反汇编代码。

```

1 | Disassembly of section .text:
2 |
3 | 0000000000000000 <main>:
4 | #include <stdio.h>
5 |
6 | int main(){
7 |     0: 55                push    %rbp
8 |     1: 48 89 e5          mov     %rsp,%rbp
9 |     4: 48 83 ec 20       sub     $0x20,%rsp
10 |     int i = 0;
11 |     8: c7 45 e8 00 00 00 00 movl    $0x0,-0x18(%rbp)
12 |     int a[] = {10,20,30};
13 |     f: c7 45 f0 0a 00 00 00 movl    $0xa,-0x10(%rbp)
14 |    16: c7 45 f4 14 00 00 00 movl    $0x14,-0xc(%rbp)
15 |    1d: c7 45 f8 1e 00 00 00 movl    $0x1e,-0x8(%rbp)
16 |     int r = 1 * a[i++] + 2 * a[i++] + 3 * a[i++];
17 |    24: 8b 45 e8          mov     -0x18(%rbp),%eax
18 |    27: 48 98            cltq
19 |    29: 8b 54 85 f0       mov     -0x10(%rbp,%rax,4),%edx
20 |    2d: 8b 45 e8          mov     -0x18(%rbp),%eax
21 |    30: 48 98            cltq
22 |    32: 8b 44 85 f0       mov     -0x10(%rbp,%rax,4),%eax
23 |    36: 01 c0            add     %eax,%eax
24 |    38: 8d 0c 02         lea     (%rdx,%rax,1),%ecx
25 |    3b: 8b 45 e8          mov     -0x18(%rbp),%eax
26 |    3e: 48 98            cltq
27 |    40: 8b 54 85 f0       mov     -0x10(%rbp,%rax,4),%edx
28 |    44: 89 d0            mov     %edx,%eax
29 |    46: 01 c0            add     %eax,%eax
30 |    48: 01 d0            add     %edx,%eax
31 |    4a: 01 c8            add     %ecx,%eax
32 |    4c: 89 45 ec          mov     %eax,-0x14(%rbp)
33 |    4f: 83 45 e8 01       addl    $0x1,-0x18(%rbp)

```

```

34 53: 83 45 e8 01      addl    $0x1, -0x18(%rbp)
35 57: 83 45 e8 01      addl    $0x1, -0x18(%rbp)
36      printf( "%d\n", r );
37 5b: 8b 45 ec          mov     -0x14(%rbp), %eax
38 5e: 89 c6            mov     %eax, %esi
39 60: bf 00 00 00 00    mov     $0x0, %edi
40 65: b8 00 00 00 00    mov     $0x0, %eax
41 6a: e8 00 00 00 00    callq   6f <main+0x6f>
42      return 0;
43 6f: b8 00 00 00 00    mov     $0x0, %eax
44 }
45 74: c9              leaveq  %eax, %edi
46 75: c3              retq

```

最先和最后的几个指令只建立了堆栈结构，初始化变量的值，调用printf函数，还从main函数返回。所以我们实际上只需要关心从0×24到0×57之间的指令。那是令人关注的行为发生的地方。让我们每次查看几个指令。

```

1 24: 8b 45 e8          mov     -0x18(%rbp), %eax
2 27: 48 98            cltq
3 29: 8b 54 85 f0       mov     -0x10(%rbp, %rax, 4), %edx

```

最先的三个指令与我们预期的一致。首先，它把i(0)的值加载到eax寄存器，带符号扩展到64位，然后加载a[0]到edx寄存器。这里的乘以1的运算（1*）显然被编译器优化后去除了，但是一切看起来都正常。接下来的几个指令开始时也大致相同。

```

1 2d: 8b 45 e8          mov     -0x18(%rbp), %eax
2 30: 48 98            cltq
3 32: 8b 44 85 f0       mov     -0x10(%rbp, %rax, 4), %eax
4 36: 01 c0            add     %eax, %eax
5 38: 8d 0c 02          lea     (%rdx, %rax, 1), %ecx

```

第一个mov指令把i的值（仍然是0）加载进eax寄存器，带符号扩展到64位，然后加载a[0]进eax寄存器。有意思的事情发生了——我们再次期待i++在这三条指令之前已经运行过了，但也许最后两条指令会用某种汇编的魔法来得到预期的结果(2*a[1])。这两条指令把eax寄存器的值自加了一次，实际上执行了2*a[0]的操作，然后把结果加到前面的计算结果上，并存进ecx寄存器。此时指令已经求得了a[0] + 2 * a[0]的值。事情开始看起来有一些奇怪了，然而再一次，也许某个编译器魔法在发生。

```

1 3b: 8b 45 e8          mov     -0x18(%rbp), %eax
2 3e: 48 98            cltq
3 40: 8b 54 85 f0       mov     -0x10(%rbp, %rax, 4), %edx
4 44: 89 d0            mov     %edx, %eax

```

接下来这些指令开始看上去相当熟悉。他们加载i的值（仍然是0），带符号扩展至64位，加载a[0]到edx寄存器，然后拷贝edx里的值到eax。嗯，好吧，让我们在多看一些：

```

1 46: 01 c0            add     %eax, %eax
2 48: 01 d0            add     %edx, %eax

```

```

3 | 4a: 01 c8          add    %ecx,%eax
4 | 4c: 89 45 ec        mov    %eax,-0x14(%rbp)

```

在这里把a[0]自加了3次，再加上之前的计算结果，然后存入到变量“r”。现在不可思议的事情——我们的变量r现在包含了a[0] + 2 * a[0] + 3 * a[0]。足够肯定的是，那就是程序的输出：“60”。但是那些后缀操作符上发生了什么？他们都在最后：

```

1 | 4f: 83 45 e8 01      addl   $0x1,-0x18(%rbp)
2 | 53: 83 45 e8 01      addl   $0x1,-0x18(%rbp)
3 | 57: 83 45 e8 01      addl   $0x1,-0x18(%rbp)

```

看上去我们编译版本的代码完全错了！为什么后缀操作符被扔到最后、所有任务已经完成之后？随着我对现实的信仰减少，我决定直接去找本源。不，不是编译器的源代码——那只是实现——我抓起了C11语言规范。

这个问题处在后缀操作符的细节。在我们的案例中，我们在单个表达式里对数组下标执行了三次后缀自增。当计算后缀操作符时，它返回变量的初始值。把新的值再分配回变量是一个副作用。结果是，那个副作用只被定义为只被付诸于各顺序点之间。参照标准的[1.2.3章](#)，那里定义了顺序点的细节。但在我们的例子中，我们的表达式展示了未定义行为。它完全取决于编译器对于什么时候给变量分配新值的副作用会执行 相对于表达式的其他部分。[↓](#)

最终，我俩都学到了一点新的C语言知识。众所周知，最好的应用是避免构造复杂的前缀后缀表达式，这就是一个关于为什么要这样的极好例子。

关于作者：[cjpan](#)



海上钢琴摄影攻城师，好吧，← 每个位面都只是刚起步而已。（新浪微博：[@潘成杰V](#)）

[查看cjpan的更多文章 >>](#)



熬过了十年寒窗
却依然每月月光
大夫说，跳跳更健康

立即访问JobDeer.com

国内第一的人才拍卖网站

相关文章

- [学习较底层编程：动手写一个C语言编译器](#)
- [C语言函数指针基础](#)
- [让C程序更高效的10个建议](#)
- [C++编译器无法捕捉到的8种错误](#)
- [C语言指针和数组基础](#)
- [很酷的C语言技巧](#)
- [为什么转置512×512矩阵，会比513×513矩阵慢很多？](#)
- [12个有趣的C语言问答](#)
- [C语言内存地址基础](#)
- [C语言指针5分钟教程](#)

« [从程序员到项目经理（18）：不要试图和下属做朋友](#)
[王远轩：北美求职记](#) »

发表评论

Comment form

Name*

姓名

邮箱*

请填写邮箱

网站 (请以 http://开头)

请填写网站地址

评论内容*

请填写评论内容

(*) 表示必填项

[提交评论](#)

15 条评论

1. *stone* 说道:

[2013/12/12 上午 10:12](#)

纠结于这些无聊的问题.写这种代码的人该枪毙.

 43  0[回复](#)2. *AntiLinuxism* 说道:[2013/12/12 上午 10:49](#)

我比较懒，不愿读spec，碰到这种情况，一般是把所有后缀++都拿到表达式最后，看来和某些编译器想到一块儿了

把后缀++写成3个放不同的地方当成一个考试题，当然是没有意义的。

但是这种规定本身是有意义的，写循环的时候，或者表示“哥已到此一游，请后来者读后面数据”的时候，有时候会方便一点，视觉上直接表达出来意图了。

一棍子打死说不让用后缀++也不行的，正宗的C程序员不会答应，呵呵

 2  0[回复](#)3. *bel* 说道:[2013/12/12 上午 11:19](#)

这也许不是什么新知识,也许换一个编译器就是另外一个结果了.这只是编译器不同实现的方式

 0  0[回复](#)4. *sui Wengerbi* 说道:[2013/12/12 上午 11:22](#)

你这解释也太麻烦了，i++;其实就是在这条“语句执行之后再给i加上的一！！

在vc上能够得出结果

版本为下面的gcc就报错le

Configured with: --prefix=/Applications/Xcode.app/Contents/Developer/usr --with-gxx-include-

dir=/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.9.sdk/usr/include/c++/4.2.1

Apple LLVM version 5.0 (clang-500.2.79) (based on LLVM 3.3svn)

Target: x86_64-apple-darwin13.0.0

Thread model: posix

 2  0

[回复](#)

◦ 沉默 说道:

[2013/12/12 下午 2:41](#)

你節約了我好多時間，謝謝你。

 0  0

[回复](#)

5. *lsldx* 说道:

[2013/12/12 下午 1:06](#)

"结果是，那个副作用只被定义为只被付诸于各顺序点之间。"

原文是:

It turns out that side effects are only defined to have been committed between sequence points.

“have been committed”是不是可以直译成“被提交”呢？

 0  0

[回复](#)

◦ *cjpan* 说道:

[2013/12/14 上午 11:26](#)

我为了防止读者被引到“代码提交”上去，所以换了个说法。

 0  0

[回复](#)

6. *Alph* 说道:

[2013/12/12 下午 10:27](#)

我用OSX编译这段代码，结果就是140.

不同的编译器有不同的实现，这根本没有什么好纠结的。关键是写出这种代码的人应该切腹谢罪。

 7  0

[回复](#)7. *wkoji* 说道:[2013/12/13 上午 5:52](#)

加个括号什么都解决了。自己研究是无所谓，假如写商业或者工业代码这么来写，直接毙掉

 4  0[回复](#)8. *alpha* 说道:[2013/12/13 下午 11:42](#)

这种问题还要纠结.....k&r里关于副作用何时生效已经说得很清楚了

 0  0[回复](#)9. *Monologue* 说道:[2013/12/15 下午 10:41](#)

这个跟编译器有关系吧？

```
int a[]={ 10,20,30,40};
```

```
int r = 1 * a[++i] + 2 * a[++i] + 3 * a[++i];
```

我在linux下直接编译结果是190,
在vs中时240,

不太明白190是怎么来的。
求解释。

```
int i=0;
```

```
8048395: c7 45 f4 00 00 00 00 movl $0x0,0xffffffff4(%ebp)
```

```
int a[]={ 10,20,30,40};
```

```
804839c: c7 45 e4 0a 00 00 00 movl $0xa,0xffffffffe4(%ebp)
```

```
80483a3: c7 45 e8 14 00 00 00 movl $0x14,0xffffffffe8(%ebp)
```

```
80483aa: c7 45 ec 1e 00 00 00 movl $0x1e,0xfffffec(%ebp)
```

```
80483b1: c7 45 f0 28 00 00 00 movl $0x28,0xfffff0(%ebp)
```

```
int r=1*a[++i]+2*a[++i]+3*a[++i];
```

```
80483b8: 83 45 f4 01 addl $0x1,0xffffffff4(%ebp)
80483bc: 8b 45 f4 mov 0xffffffff4(%ebp),%eax
80483bf: 8b 4c 85 e4 mov 0xffffffe4(%ebp,%eax,4),%ecx
80483c3: 83 45 f4 01 addl $0x1,0xffffffff4(%ebp)
80483c7: 8b 45 f4 mov 0xffffffff4(%ebp),%eax
80483ca: 8b 54 85 e4 mov 0xffffffe4(%ebp,%eax,4),%edx
80483ce: 89 d0 mov %edx,%eax
80483d0: 01 c0 add %eax,%eax
80483d2: 8d 14 10 lea (%eax,%edx,1),%edx
80483d5: 83 45 f4 01 addl $0x1,0xffffffff4(%ebp)
80483d9: 8b 45 f4 mov 0xffffffff4(%ebp),%eax
80483dc: 8b 44 85 e4 mov 0xffffffe4(%ebp,%eax,4),%eax
80483e0: 01 c0 add %eax,%eax
80483e2: 8d 04 02 lea (%edx,%eax,1),%eax
80483e5: 8d 04 01 lea (%ecx,%eax,1),%eax
80483e8: 89 45 f8 mov %eax,0xffffffff8(%ebp)
printf("-----%d-----\n",r);
```

 0  0

[回复](#)

10. *Edward Shen* 说道:

[2013/12/16 下午 2:04](#)

可以这么写代码吗？在一个表达式里多次进行++/--这样的操作。其顺序是未定义的。这种代码是不合格的。其结果应该对于不同的编译器有不同的输出！根本不值得讨论！

 0  0

[回复](#)

11. *marco* 说道:

[2013/12/19 上午 11:30](#)

纠结这种无聊的问题.还不如看下代码规范或者人家写的东西.而且文章内容对不住标题...优秀的代码规范比纠结这样的问题要好得多...

这个文章唯一告诉我们的就是:

一个优秀的c码农应该在代码里尽量避免诸如此类能让编译器误会的问题...

面试的时候谁写这样的代码就TM直接让他回炉...

 3  0

[回复](#)

12. *bombless* 说道:

[2014/07/11 下午 3:57](#)

这个规则很有意思，不是完全没有用的。

看上去就是C++的临时对象何时析构的翻版：临时对象在最外层表达式的最后析构，也就是往外搜索，直到表达式不再是另一个表达式的一部分。

因此这个部分不只是有趣的知识，它是应该被准确掌握的。

 0  0

[回复](#)

Search for:



[伯乐头条 – 分享和发现原创](#)

为作者带来更多读者；为读者筛选优质内容；专注IT互联网。

极客学院
jikexueyuan.com



Java语言学习极速之旅

只用2个月,你系好安全带了吗?

立即开始学习

- [本周热门文章](#)
- [本月热门文章](#)
- [热门标签](#)

0 [ASP.NET应用程序和页面生命周期](#)

1 [揭开计算机的神秘面纱](#)

2 [开始学习Linux的一些建议](#)

3 [在 Linux 下你所不知道的 df 命令的那些功能](#)

4 [网站静态化处理—前后端分离\(下\)](#)

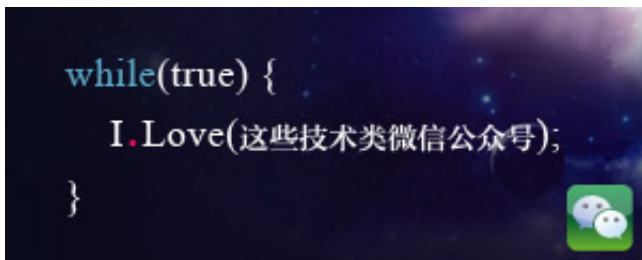
5 [网站静态化处理—前后端分离\(中\)](#)

6 [C 语言中的指针和内存泄漏](#)








7 [C#中的Lambda表达式和表达式树](#)

8 [卓越程序员和优秀程序员有哪些区别？](#)

9 [网站静态化处理—前后端分离\(上\)](#)



最新评论（期待您也参与评论）

-  Re: [如果看了此文你还不理解傅里叶变换，那就过来掐死我吧【完整版】](#)
写的真好。牛，大赞一个 [林乔扬](#)
-  Re: [如何在Linux命令行中创建以及展示演示稿](#)
这个东西，估计到死也用不上一回 test
-  Re: [为什么 Vim 使用 HJKL 键作为方向键](#)
这个可以换成esc or ctrl caimaoy
-  Re: [Objective-C Runtime 运行时之二：成员变量与属性](#)
//补全最后一个例子-(NSString*)propertyForKey:(NSString*)ke... reader
-  Re: [Objective-C Runtime 运行时之二：成员变量与属性](#)
最后一个例子，怎么没看到map的使用呢？ reader
-  Re: [Objective-C Runtime 运行时之二：成员变量与属性](#)
动态地增强类现有的功能，当然你也可以使用分类、扩展、派生新类实现，但这就是静态扩展了。 reader
-  Re: [通俗解释「为什么数据库难以拓展」](#)

文章写得很不错，通俗易懂 [xinwendashibaike](#)



•

Re: [C++开发者都应该使用的10个C++11特性](#)

std::shared_ptr(new int(42)), seed())对于函数而言, 它的参数的... ss

热点频道



[Python开发频道](#)

汇集优质的Python技术文章和资源。人生苦短，我用Python！



[前端开发频道](#)

JavaScript, CSS, HTML5 这里有前端的技术干货！



[安卓开发频道](#)

关注安卓移动开发业界动态，分享技术文章和优秀工具资源。



[iOS开发频道](#)

关注iOS移动开发业界动态，分享技术文章和优秀工具资源。

推荐关注



[伯乐头条](#)

为作者带来更多读者；为读者筛选优质内容；专注IT互联网。



[伯乐翻译小组](#)

由数百名译者组成，立志翻译传播优秀的外文技术干货。



[程序员相亲](#)

一个专门为IT单身男女服务的征婚传播平台。



[资源导航](#)

收录优秀的工具资源，覆盖开发、设计、产品和管理等。



关于伯乐在线博客

在这个信息爆炸的时代，人们已然被大量、快速并且简短的信息所包围。然而，我们相信：过多“快餐”式的阅读只会令人“虚胖”，缺乏实质的内涵。伯乐在线博客团队正试图以我们微薄的力量，把优秀的原创/译文分享给读者，做一个小而精的精选博客，为“快餐”添加一些“营养”元素。



欢迎关注更多频道

[头条](#) – 分享和发现有价值的内容与观点
[相亲](#) – 为IT单身男女服务的征婚传播平台
[资源](#) – 优秀的工具资源导航
[翻译](#) – 翻译传播优秀的外文文章
[博客](#) – 国内外的精选博客文章
[前端](#) – JavaScript, HTML5, CSS
[安卓](#) – 专注Android技术分享

[iOS](#) – 专注iOS技术分享
[Java](#) – 专注Java技术分享
[Python](#) – 专注Python技术分享

联系我们

商务合作

Email: [bd@Jobbole.com](mailto:bd@jobbole.com)
QQ: 2302462408 (加好友请注明来意)

网站使用问题

请直接[联系我们](#)询问或者反馈

© 2015 伯乐在线 [头条](#) [博客](#) [资源](#) [翻译](#) [小组](#) [相亲](#)

本站由 [UCloud](#) 赞助云主机, [七生](#) 赞助云存储