Into the Depths of C: Elaborating the De Facto Standards
调研报告

PB15111616

林鑫

摘　要

C 语言规范被 ISO 标准定义，但实际中 C 语言的实现与存在的代码的对 C 语言的理解存在歧义，与 ISO 标准不同。文章中作者通过问卷调查调研实际应用中 C 语言的不同参与者对 C 语言的理解，对 C 语言中指针与内存操作的语义歧义进行详细讨论。同时作者提出一种 C 语言的规范化模型 Cerberus，通过对底层内存模型的修改，可以应用于不同的规范化模型，可作为小规模样例的测试集。

关键词：C　指针　内存模型

目 录

摘要 ·····

第 1 章 引言 ····· 2

第 2 章 指针与内存歧义 ····· 4

2.1 指针来源 ····· 4

2.2 指针越界 ····· 5

2.3 指针拷贝 ····· 6

2.4 不确定值 ····· 6

2.5 不确定填充位 ····· 6

第 3 章 Cerberus ····· 7

3.1 Cerberus 架构 ····· 7

3.2 内核概述 ····· 8

3.3 未定义行为 ····· 9

3.4 算术运算 ····· 9

3.5 次序模型 ····· 9

3.6 生存期 ····· 10

3.7 循环和跳转 ····· 11

3.8 内存模型 ····· 11

第 4 章 总结 ····· 13

参考文献 ····· 14

第 1 章　引言

C 语言是 40 多年前开发的，现在仍然是大多数软件系统的开发语言，被广泛应用于系统编程。尽管 C 语言规范被 ANSI/ISO 标准定义，诸如 C89/C90，C99 和 C11 规范，但由于 C 语言的开发和使用都较早，不同公司的 C 编译器的实现不同，对 C 语言规范的支持也各不相同，而且开发者对 C 语言的理解也不尽相同，因此实际中真正被使用的 C 语言规范十分复杂，C 语言的规范化存在大量问题。

C 语言的不同参与者对 C 行为的假设存在歧义。在 C 语言的实际使用中，程序员对 C 编译器行为的假设、不同 C 编译器的实际行为、已存在的 C 代码对 C 语言的假设、C 分析工具对 C 语言的假设往往各不相同，不同开发者与使用者所认为的 C 语言规范各不相同，与 ISO 标准也不同。这些差异并不只是理论上的问题，在 C 语言的实际使用中也会带来许多困境。例如，现代的 C 编译器在代码优化上越来越激进，代码优化依赖于编译器对 C 行为的假设，所利用的一些情况可能被 C 规范视为未定义行为，已经存在的 C 代码适用于过去对 C 行为的假设，却未必符合现在对 C 的假设，可能存在一些现在编译器不再支持的用法，因此过去可以正常运行的代码使用现在的编译器重现编译后可能会出现运行出错或者存在一些安全问题，这种现象在许多关键的代码如操作系统内核中大量存在，存在巨大的隐患，编译时往往需要使用特定的编译选项控制编译器的行为，使其行为符合过去的假设，但这些在 ISO 规范中并没有提到。C 语言分析工具为了配合主流的 C 语言假设，往往会使用特殊的操作语义以避免出现大量用户所不期望的错误，而这些特殊的语义可能与 ISO 规范相悖。考虑到 C 编译器实际实现中需要考虑的一些问题，实际 C 编译器的实现往往比 ISO 规范中所描述的简单，ISO 规范制定时并没有考虑到实际硬件的一些情况。

C 语言不存在准确可用的规范。C 语言尝试在使程序员能够写出高效、可手写的底层代码的同时，支持大量的不同目标机器架构和复杂的编译器优化。C 同时提供了在抽象值与具体的底层表示上的操作来实现这个目的，但两者的映射关系十分复杂，涉及内存和类型安全、重命名、并发等问题，在标准的规范式描述下很难准确描述，标准的准确翻译是什么存在很大争议，作为标准的规范也无

法作为可执行的测试集检测编译器的行为是否正确，因此不同编译器对标准的理解与实现存在很大差异。而解决这一问题的一种思路是使用数学形式化描述规范，但这一方法忽略了实际使用的 C 规范，实际上作为开发者与用户之间的接口应该同时对两者可用，数学化的描述虽准确但对开发者却是不可用的。

C11 的并发与序列语义与其余内容的形式统一存在问题。C11 标准正式支持并发，但 C11 标准中并发模型使用公理化描述，而其余部分使用自然语言化描述操作语义，这两者显然无法很好的统一。

总而言之，现在 C 语言规范的状态时一片混乱。ISO 标准与实际使用的规范的歧义，ISO 标准的模糊描述，并发模型的描述形式无法统一都使得 ISO 标准无法提供一个令人满意的 C 语言定义。

第 2 章　指针与内存歧义

作者通过问卷调查调研实际代码中所使用的主流的 C 语义，问卷通过一些用法在常见 C 编译器中是否能运行以及在实际中是否被使用的方式获取开发者对其所理解的 C 语义的假设，收到的回复包括 C 应用程序开发者，C 系统开发者，操作系统开发者，C 嵌入式系统开发者，C 标准制定者，编译器开发者和分析工具开发者等 C 语言的各方参与者，调研的问题主要包括指针来源，指针比较，指针算术运算，指针强制类型转换，结构体和联合体类型访问，指针生存期，无效访问，陷阱表示，不确定值，结构体与联合体填充，有效类型等方面。根据调研结果，开发者，用户和标准对 C 的指针和内存模型理解存在歧义，在调研的 85 个问题中，有 39 个问题 ISO 标准不清晰，有 27 个问题 C 应用标准不清晰，甚至在一些情况下，C 的使用与实现之间存在巨大差异，有 27 个问题 ISO 标准与 C 应用标准有歧义。下面列举一些指针与内存相关的歧义。

2.1　指针来源

ISO DR260 委员会认为“在确定操作是否被定义时需要考虑指针的来源”。以以下程序为例：

```
1  #include <stdio.h>
2  #include <string.h>
3  int y = 2, x = 1;
4  int main(){
5        int *p = &x + 1;
6        int *q = &y;
7        printf("Addresses:␣p=%p␣q=%p\n",(void*)p, (void*)q);
8        if(memcmp(&p, &q, sizeof(p)) == 0){
9              *p = 11;
10             printf("x=%d␣y=%d␣*p=%d␣*q=%d\n", x, y, *p, *q);
11       }
12       return 0;
13 }
```

在不考虑指针来源的情况下，如果 x 和 y 分配在相邻的的内存中，那么 &x+1 和 &y 按位将完全相同，memcmp 将返回 0，指针变量 p 和 q 都将指向 y，期待的结果应该是 x=1 y=11 *p=11 *q=11。但 GCC 返回的结果是 x=1 y=2 *p=11 *q=2，说明 GCC 考了虑指针来源，不认为 *p 与 y 或 *q 相同，初值 y=2 会传播到最后

的 printf 函数。

DR260 认为指针变量不应该只包含具体地址，也应该包含指针的来源信息。指针的的来源信息可以在内存访问时判断使用的地址是否与分配的地址一致。在该语义下，上述程序片段中的 *p=11 操作将会被视为未定义的行为，程序被认为是错误的，理论上编译器为了分析与优化可以任意处理这种情况。一般来说，出现未定义行为的情况需要程序员避免，从而使编译器为了便于优化和分析可以作出一些强假设。指针变量携带来源信息在 C 以及一些类 C 语言的实现中十分普遍，但它会带来许多问题。

例如，在分别独立分配的指针间是否可以进行 <，>，<=，>= 等关系运算的问题上，ISO 标准明确指出这一行为是禁止的，但在实际中确是广泛使用的，因此主流的 C 语义应该接受这一行为而不是严格按照 ISO 标准。

在是否可以通过指针加减构造出一个可用的地址的问题上，C 的使用与 ISO 标准及编译器实现间存在歧义，在一些编译器的实现中，编译器在优化时假设不会存在这种用法，但在一些重要场合，如操作系统的内核代码中，这种用法确实存在。一些可能的解决方法都存在性能损失或代价太大的问题，这一问题尚不存在令人满意的解决方法。

在指针强制类型转换为整数或进行整数运算时是否需要记录来源信息的问题上，GCC 文档显示在将指针强制类型转换为整数，再转换回指针时，指针携带的来源信息必须与原来指针携带的来源信息一致，但如果不存在所谓的原来的指针时该如何处理存在问题。

在指针相等判断结果是否会受到来源信息影响的问题上，ISO 标准允许在不同来源的指针上进行相等判断，但并未规定判断结果是否应该受到来源信息的影响。

2.2 指针越界

在 ISO 标准中只有少数的指针运算是被允许的，尤其是在数组，结构体成员，对象传递时。但在实际应用中，经常会出现构造一个临时越界指针的情况。在调查中，大多数人认为只要在使用指针访问内存之前确保指针没有越界，则程序可以正常运行，但编译器在进行优化时往往假设这种情况不会存在。

2.3 指针拷贝

在指针携带来源信息的前提下，库函数 memcpy 在拷贝指针时肯定会同时复制指针的来源信息，但是用户代码中复制指针的二进制数值时是否会同时复制指针的来源信息，对于这一问题大多数人都持肯定态度，但实际编译器实现是否支持仍需验证。

2.4 不确定值

C 语言允许变量不初始化，读取一个未初始化的变量的结果存在多种语义：未定义行为，编译器可以任意处理；或使包含该变量的表达式的结果不可预测；或返回一个随机而不稳定的值，再次读取时结果不同；或返回一个随机但稳定的值，再次读取时结果相同。这种情况会在复制一个部分初始化的结构体时出现，出于不同的考虑，不同编译器的实现不同。在该问题上，ISO 标准的规定趋于混乱，而且不符合主流的 C 语义。

2.5 不确定填充位

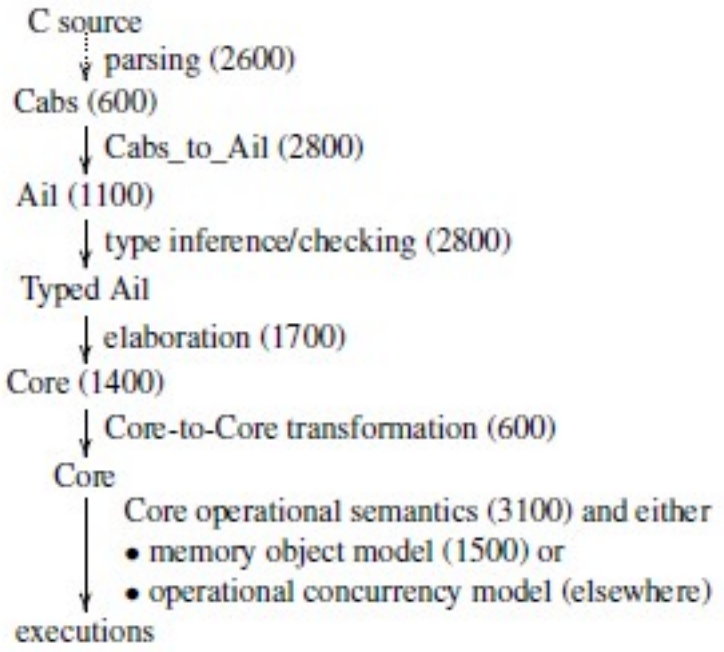
不确定值的问题在填充位中同样存在，填充位同样存在多种语义：无论被写入任何内容，填充位都是不确定值；或写入结构体成员时同时在填充位写入不确定值；或写入 0 或者不修改；或结构体复制同时复制填充位，但写入结构体成员不修改填充位。在这个问题上的调查结果多种多样，需要一种机制可以确保填充位不会引发信息泄露等安全问题。

第 3 章　Cerberus

作者为 C 的大部分内容提出一种规范化模型 Cerberus。Cerberus 的内存模型可以参数化，使其可以使用应用 C 内存模型，也可以使用 C11 并发模型。此外，Cerberus 作为可执行程序可以用作小规模样例的测试集。

3.1　Cerberus 架构

Cerberus 的架构如下图所示：



编程语言的动态语义一般使用与源语言类似的抽象语法类型 AST 描述，由于 C 语言中的一部分操作语义与编译时的一些问题相关，为了降低处理的复杂度，Cerberus 没有直接采用类 C 的抽象语法类型，而是经过一系列处理将类 C 的 AST 翻译为更小更简单的内核语言。为适应应用 C 规范而对动态语义的修改只会影响类 C 抽象语法类型的翻译过程，而内核语言保持不变。

Cerberus 的处理过程如下：在常规的 C 预处理之后，前端使用 Menhir 语法分析器，严格按照 ISO 语法产生 AST Cabs；经过 Cabs_to_Ail 过程生成一个更简洁的 AST Ail，方便之后的类型检查与内核实现，这个过程包括变量作用域确定，

函数原型声明与函数定义，规范化 C 的词法，字符串与枚举类型处理，规范化循环为 while 循环等，处理过程严格按照标准进行，如果由于程序问题导致过程失败，将会准确指出所违背的标准；类型检查给出显式的类型声明，同样如果出错将精确给出违背的标准；经过可选的 Core-to-Core 简化过程之后，驱动器将内核与一个内存模型连接，内存模型可能是应用 C 内存模型或 C11 并发模型。

3.2 内核概述

内核语言是一个类型化的按值调用语言，由函数定义和表达式构成，一阶公民包括递归函数，列表，元组，布尔值，整数值，C 指针类型，C 函数指针类型，以及用于表示类 C 的 AST 项的 ctype 类型。内核语言中行为尽可能显式指明以避免模糊的语义。

bTy ::=	**Core base types**
`unit`	unit
`boolean`	boolean
`ctype`	Core type of C type exprs
[*bTy*]	list
$(\overline{bTy_i}^{\,i})$	tuple
oTy	C object value
`loaded` *oTy*	*oTy* or unspecified value

value ::=	**Core values**
object_value	C object value
`Specified` (*object_value*)	non-unspecified loaded value
`Unspecified` (*ctype*)	unspecified loaded value
`Unit`	unit
`True`	true
`False`	false
ctype	C type expr as value
$bTy[value_1, .., value_n]$	list
$(value_1, .., value_n)$	tuple

内核语言中标识符在某个值被生成时与值进行绑定，而不是类 C 的可修改变量，与内存对象的交互通过内存操作 create，kill，load，store，rmw 进行。

3.3 未定义行为

Cerberus 明确指出程序中存在的未定义行为，未定义行为由 undef 构造，可能由两种方式动态生成：C 算术运算中存在未定义行为或者内存访问（不安全内存访问，数据竞争等）。对于前者，在执行算术运算时对操作数进行显式检查，如果算术运算的操作语义触发了未定义行为，内核停止运行，并报告发生了哪种未定义行为，所有执行路径上的存在的未定义行为都会被检查到；对于后者，内存模型将会使用 sequenced-before 或 happens-before 模型检测出未定义行为。

3.4 算术运算

C 语言中的许多整数类型由于取值范围，是否包含符号位，不同的表示长度，对齐限制，隐式类型转换等问题会引发大量编译错误和异常执行结果，而在 Cerberus 中这些问题都会被类型推断过程检测出，在内核中仅进行操作数检查以及简单的算术运算。

3.5 次序模型

C 标准定义了表达式的求值次序，但 C 规范对于求值次序的定义是一个松散的定义，表达式的求值过程中各操作之间既有严格的先后关系，也有不明显的次序关系。为表示 C 的这种复杂次序关系，Cerberus 的内核引入以下次序模型：

$e ::= \ldots$	
$\mid$ unseq $(e_1, .., e_n)$	unsequenced expressions
$\mid$ let weak $pat = e_1$ in e_2	weak sequencing
$\mid$ let strong $pat = e_1$ in e_2	strong sequencing
$\mid$ let atomic $(sym : oTy) = a_1$ in pa_2	atomic sequencing
$\mid$ indet $[n]\,(e)$	indeterminate sequencing
$\mid$ bound $[n]\,(e)$	. . .and boundary
$\mid$ nd $(e_1, .., e_n)$	nondeterministic seq.

unseq 允许其包含的任意一个表达式 e_i 求值时在满足其自身的 sequenced-before 约束条件下，可以与所包含的其它表达式求值时的内存操作任意交叉，返回一个表达式结果组成的元组。

let weak 和 let strong 将 e_1 的结果绑定于 pat，并明确定义了求值 e_1 的内存操作在求值 e_2 的内存操作之前。考虑 C 的后置自增和自减运算符，赋值表达式

右值的求值必须先于自增自减结果的保存，直观上看，这是影响最终结果的求值过程的一部分，但在包含自增自减运算的表达式中，表达式的求值过程实际上排除了自增自减结果保存的过程。为了在内核中表示 C 的这一特性，给内存操作引入极性，不是表达式求值的一部分的内存操作被表示为负性。强次序关系确保 e_1 的所有内存操作先于 e_2 进行，不考虑内存操作的极性，而弱次序关系考虑极性，只确保 e_1 中的正性内存操作先于 e_2 进行。

$pa ::=$ **memory actions with polarity**
| a positive, sequenced by both `let weak` and `let strong`
| `neg` (a) negative, only sequenced by `let strong`

lei atomic 表示原子操作，确保内存操作 a_1 在 pa_2 之前执行，而且不会有其他内存操作在 a_1 与 pa_2 之间执行；

indet 和 bound 用于表示表达式中非确定性的次序关系的情形，indet 用于描述一个子表达式在考虑上下文的情况下可以以某一非确定性的次序执行，而 bound 指明了相关上下文的部分。Core-to-Core 转换通过以上操作重写程序，将所有表达式转换重写为没有非确定性次序关系的操作序列，indet 和 bound 不会在该操作序列中出现。

3.6 生存期

C 语言中一般通过作用域和动态分配来确定对象的生存期，而 Cerberus 中没有块的概念，所有对象的生存期通过显式的内存构造与删除操作显式指明。Cerberus 通过 ctype 类型记录每个 C 类型的大小等相关信息，内存构造操作接受对齐约束与分配大小或 ctype 作为操作参数，同时对内存对象的 load、store 或 rmw 操作接受左值的 ctype 作为参数，以此描述 C 语言中内存操作的约束。

$a ::=$ **memory actions**
| `create` (pe_1, pe_2)
| `alloc` (pe_1, pe_2)
| `kill` (pe)
| `store` $(pe_1, pe_2, pe, \mathit{memory\text{-}order})$
| `load` $(pe_1, pe_2, \mathit{memory\text{-}order})$
| `rmw` $(pe_1, pe_2, pe_3, pe_4, \mathit{memory\text{-}order}_1, \mathit{memory\text{-}order}_2)$

3.7 循环和跳转

Cerberus 中使用 continuation 实现 C 语言中的 for，while，do 循环及相关的 break，continue，exit 等语义，continuation 通过 save，run 和 skip 操作，这种处理方式可以保留程序原有基本结构，以如以下程序片段所示：

```
while(e){
      s1;
      break;
      s2;
}
```

```
save l() in
      let strong id=[e] in
      if id=0 then
            save b() in skip
      else
            let strong _ = [s1] in
            let strong _ = run b() in
            let strong _ = [s2] in
            run l()
```

对于 C 语言中的 goto 跳转，当跳转进程序块时，run 操作会检查 save 目标中所需要的所有对象，并创造所需的对象，当从程序块中跳出时，run 操作相应删除所有不再需要的对象。

3.8 内存模型

考虑上文中关于指针来源信息的讨论，需要对指针以及整数值进行重新定义。在应用 C 内存模型中，指针和整数值都带有来源信息，对于 NULL 指针或纯整数值，来源信息为空，对于指针来源信息继承自原值携带的对象最初分配的 ID，或来自 IO 的通配符。对于大多数包含来源信息的值与纯数值的算术运算将会保留来源信息，但两个值的减法将会产生一个纯数值，而两个包含不同来源信息的值的算术运算同样生成一个纯数值。

指针还包含一个对象分配时的基础 ID，或从整数的基础转换信息，以及一个偏移用于数组或成员变量访问。

整数值是具体数值，内存对象地址，偏移，或指针转换的数值，指针差值，对象大小等。

内存值可能为未确定值，整数值，指针，数组，联合体，结构体。

对于上文指针来源中提到的程序示例，x 和 y 在分配时将会生成关联的 ID，&x 和 &y 将会保留 x 和 y 的来源，&x+1 将会保留 x 的来源信息，memcmp 只比

较指针二进制表示，不考虑指针来源信息。在 *p=11 访问中，将会比较指针地址与来源信息是否匹配，在这里明显不匹配，因此动态语义将会报告未定义行为。

第 4 章　总结

在过去的 40 年中，C 语义的解释成为一个复杂的问题。编译器优化，已存在的系统代码，分析工具之间的交互以及安全问题使得这个问题更加重要。作者关于 C 应用标准与 ISO 标准的各个层面的深入调查，可以帮助梳理目前的情形，对于未来的测试、语义、分析、验证与标准化工作有很大帮助。作者提出的 Cerberus 规范化模型描述了大部分 ISO 标准，通过对底层内存模型的修改，Cerberus 可以适应多种规范化模型，作为一个可执行的小规模测试集，Cerberus 有助于未来的规范化工作。

参考文献

[1] [PLDI 2016] Into the Depths of C: Elaborating the De Facto Standards