How ISO C became unusable for operating systems development

Victor Yodaiken
vy@e27182.com
E27182
Austin, Texas, USA

Figure 1. Ken Thompson and Dennis Ritchie at Bell Labs

Abstract

The C programming language was developed in the 1970s as a fairly unconventional systems and operating systems development tool, but has, through the course of the ISO Standards process, added many attributes of more conventional programming languages and become less suitable for operating systems development. Operating system programming continues to be done in non-ISO dialects of C. The differences provide a glimpse of operating system requirements for programming languages.

Keywords: operating systems, programming languages, C, UNIX, ISO-C, compiler

1 Introduction

The C programming language [33] is the first, and, so far, only widely successful programming language that provides operating system developers with a high-level language alternative to assembler (compare to [42]). C's success was predicated on its design: a small language, close to the machine yet with a great deal of flexibility for experienced programmers. The Rationale for the C standard [9] cited C's capability to function as a "high-level assembler" and explained that *"many operations are defined to be how the target machine's hardware does it rather than by a general abstract rule"* but C also has traditional attributes of an ALGOL style programming language.

At present most major commercial operating system kernels and many experimental ones are written primarily in C or some dialect. However, ISO C [8, 26], the language that has evolved over nearly forty years of the standards process, has not only diverged from Kernighan and Ritchie C (K&R C) [13], but has become poorly suited to operating systems development. C based operating systems projects on ISO C-compliant compilers [27, 28] rely on compiler specific opt-outs, assembler escapes, and coding tricks to produce a

usable C dialect, largely undermining the purpose of an ISO standard. Among the techniques used in Linux are multiple gcc-specific opt-outs ([40] fig. 9) and "opaque" operations such as in-line assembler code that hides some pointer addition for per-CPU data structures [36]. These techniques are not necessarily stable or reliable as the compilers evolve and change C semantics.

A common argument (made e.g. by Dietz [1]) is that programmers are wrong: their objections to changes in C semantics embody "a fundamental and pervasive misunderstanding: the compiler [is] not 'reinterpreting' the semantics but rather [is] beginning to take advantage of leeway explicitly provided by the C standard." However, that overlooks the rationale's claimed intention to continue supporting the high-level assembler use case.

Limitations of ISO C for OS development have been noted in academic literature

> "*Systems or library C codes often cannot be written in standard-conformant C*" [20].

and by practitioners e.g. [38]. The primary cause is a design approach in the ISO standard that has given priority to certain kinds of optimization over both correctness and the "high-level assembler" [9] intentions of C, even while the latter remain enshrined in the rationale.

For example, a well-known security issue in the Linux kernel was produced by a compiler incorrectly assuming a pointer null check was unnecessary ([40] fig. 6) and deleting it as an optimization. Or consider this (simplified) patch report for Linux [25]:

> The test [for termination] in this loop: [...] was getting completely compiled out by my gcc, 7.0.0 20160520. The result was that the loop was going beyond the end of the [...] array and giving me a page fault [...]
> I strongly suspect it's because `__start_fw` and `__end_fw` are both declared as (separate) arrays, and so gcc concludes that `__start_fw` can never point to `__end_fw`.
> By changing these variables from arrays to pointers, gcc can no longer assume that these are separate arrays.

Here, gcc is "optimizing" based on the assumption that one externally defined object cannot overlap another, even though the host platform allows this.

ISO delegates to the compiler a great deal of the control that K&R C divides between the programmer, the environment, and the target architecture but *not* the compiler, as witnessed by the rationale's reference to "how the target machine does it". Consider a simple scalar read `*x = y`. In K&R C, a programmer could be reasonably certain that code would be generated to copy a value from the storage starting at the location of y to the location with address x, something like `load x,Reg1; load (y),Reg2; store Reg2,(Reg1)`

or perhaps simpler if some values are already cached in registers. In ISO C, things are complicated. The compiler might generate no code at all, if, for example it detects the address in x to be aliasing a storage location that is used elsewhere with a different enough type. Or the compiler might conclude that y is uninitialized and generate code to copy some arbitrary value to `*x` on the assumption that any arbitrary value will do, perhaps a different one on each access. The compiler is often free to reorder statements if it detects or assumes they do not have data dependencies.

For an example of an implementation of `malloc` in K&R (page 187) the text explains there is a question about whether "pointers to different blocks ... can be meaningfully compared", something not guaranteed by the standard. The conclusion is "this version of `malloc` is portable only among machines for which general pointer comparison is meaningful." – delegating the semantics to the processor architecture. There is no suggestion that the code is invalid. In contrast, consider the much higher-level, compiler-controlled view of comparing pointer equality in [5] (my bold).

> A priori, pointer equality comparison (with == or !=) might be expected to just compare their numeric addresses, but we observe gcc 8.1 -O2 sometimes regarding two pointers with the same address but different provenance as nonequal. Unsurprisingly, this happens in some circumstances but not others, e.g. if the test is pulled into a simple separate function, but not if in a separate compilation unit. To be conservative w.r.t. current compiler behaviour, pointer equality in the semantics should give false if the addresses are not equal, but **nondeterministically** (at each run-time occurrence) either take provenance into account or not if the addresses are equal – this specification looseness accommodating implementation variation.

The ISO approach of delegating to the compiler, not the machine, has damaging effects on reliability and expressive range for OS programming. The next two sections discuss further examples of this. Since compiler optimizations are the reason behind this approach, the final section discusses reasons for skepticism about whether this is necessary or helpful in this specific domain. Whether these kinds of optimizations are necessary for other types of applications is not within scope.

Related Work and Scope. There have been a number of articles and essays on the controversial "undefined behavior" mechanism employed to enable many ISO C optimizations (e.g., [2, 15, 30, 40, 41]): the effects on operating system programming are discussed in section 3. The complexity of ISO C semantics has also motivated development of formalizations intended to be easier to reason about, more precisely specified, or more useful for low-level programming [6, 11, 20]

and extensions to compiler intermediate languages *e.g.* Lee [17]. These are clearly related works, as discussed below, but they assume exactly what is being questioned here: the utility of the optimization approach of ISO C. The CompCert compiler [18] is discussed in the final section.

2 Optimization and time bombs

Dennis Ritchie [32] wrote the following as part of an objection to one of the first ANSI C standard drafts:

> The fundamental problem is that it is not possible to write real programs using the X3J11 definition of C. The committee has created an unreal language that no one can or will actually use.

Ritchie's main objection was to a type attribute intended to limit aliasing (two or more active pointers/references addressing the same storage).

> the committee is planting timebombs that are sure to explode in people's faces. Assigning an ordinary pointer to a pointer to a 'noalias' object is a license for the compiler to undertake aggressive optimizations that are completely legal by the committee's rules, but make hash of apparently safe programs.

C's pointer system can be a problem for optimizing compilers. Even for something as apparently innocent as

```
for(i=0; i < *b; i++)a[i] = a[i]+ *v;
```

if it is possible that for some i, a+i == v or a+i == b or even a+i = &v so compiled code must reload both values on each iteration of the loop[1]. In theory, the more the compiler can restrict aliasing, the more it can optimize. Stronger alias rules should permit more common subexpression elimination and redundancy elimination. Ritchie was dubious:

> Perhaps there is some reason to provide a mechanism for asserting, in a particular patch of code, that the compiler is free to make optimistic assumptions about the kinds of aliasing that can occur. I don't know any acceptable way of changing the language specification to express the possibility of this kind of optimization, and I don't know how much performance improvement is likely to result.

The ANSI Committee, soon to become the ISO Committee, backed down in the face of Ritchie's objections — temporarily. A year later, C89 imposed type restrictions on access to C objects as a way of facilitating type-based alias analysis (TBAA) (section 3.3 in C89). The basic idea is that ISO C forbade accessing an object of one type via a pointer (or other "left hand side") of a different enough type, with an exception for character pointers which can access everything (sort of – as discussed below).

---

[1]Here I am treating pointer equality in terms of machine addresses.

The usual example of TBAA optimization involves "lifting" variables out of loops. For the loop above

```
long  b1 = *b; long  v1 = *v; //lifted
for(int i= 0; i < b1; i++)a[i]= a[i]+v1);
```

is a legal optimization if the type of a[i] doesn't match the types of *b and *v because the compiler "knows" that those pointers cannot alias objects of a different type.

The standard does not require compilers to flag aliasing violations or to prove the absence of aliasing. Instead the ISO standard permits compilers to *assume* an absence of aliasing in a number of cases including cases where there would otherwise be type violations. Consider the following code fragment [44] where a floating point value is set to −3.14 and then an aliasing unsigned int pointer is used to turn off the sign bit. With no optimization, under Clang 12.01, this program prints 3.14. With optimization level 2, it prints −3.14 because, assuming the pointer cannot alias the floating point variable, the compiler discards the mask operation as an optimization.

```
float  f; long *l = (long *)&f;
f= -3.14;
//forbidden aliasing
*l  &= 0x7fffffff;
printf("f = %f \n", f); return;
```

Since the assumption that there is no aliasing is false in this case, a programming type error is silently "optimized" to a logic error.

Under these rules, radix sort is only permitted for elements that have byte-sized radixes.

2.1 Effects on Operating System development

For operating systems the effects are widespread particularly for objects that have different semantics depending on which kernel component is accessing them. A single block of storage may be addressed by the disk manager as a block of unsigned characters, by the file manager as an array of inodes, a directory block, or a block of characters, and by a page manager via a void pointer, all at the same time. The aliasing rules of ISO C are not compatible with this approach. It may be possible in ISO C to push all these different types into a union, but that would harm modularity, by requiring each of the components to share the basic data structures of the others.

As another example, computing a checksum for a data structure by aliasing it with an int pointer is not permitted.

```
packet_t * p = getpacket();
int ck=0;
int i;
int *q = (int *)p;  //cast is ok
for(i= 0; i< sizeof(packet_t)/sizeof(int); i++)
         ck ^= q[i]; //not permitted
```

There is no general escape mechanism, even though char pointers are allowed to alias anything[2]. The absence of escapes is a major change in C's type system as Brian Kernighan's critique of Pascal makes clear [12]:

> There is no way [in Pascal] to override the type mechanism when necessary, nothing analogous to the "cast" mechanism in C. This means that it is not possible to write programs like storage allocators or I/O systems in Pascal, because there is no way to talk about the type of object that they return, and no way to force such objects into an arbitrary type for another use.

As we saw earlier, the implementation of `malloc/free` in K&R is not conformant ISO C code. ISO C needs special rules around "effective types" (explored later) in order to permit `malloc/free` to work. Even then, it remains unclear how to write these functions in conformant ISO C.

Perhaps the most important effect is the loss of semantic clarity. Programmers are basically mystified by the rules: see "The Strict Aliasing Situation is Pretty Bad" [31] (the comments are especially illuminating). Thirty years after the aliasing type restrictions went into the Standard, a committee of standards experts wrote that the current situation "leaves many specific questions unclear: it is ambiguous whether some programming idioms are allowed or not, and exactly what compiler alias analysis and optimisation are allowed to do." [5].

In practice, alias analysis in gcc and Clang has unpredictable effects. Gcc does not omit the mask operation in the floating point example above although it does do typed alias optimizations sometimes (see Figure 7 in [40] for an example). Linux uses a flag to disable "strict-aliasing" analysis in gcc [37, 38] but that does not disable all alias optimizations as shown by an example in [5] (page 15). And because aliasing violations may not be flagged, there can be silent, surprising, changes in code operation between optimization levels or versions. This is not a C-specific problem. Fortran has similar problems with similar assumptions: "anything may happen: the program may appear to run normally, or produce incorrect answers, or behave unpredictably." [24].

Violation of aliasing rules is just one example of a large class of "undefined behaviors" and the next section looks at that topic more generally.

3 Undefined behavior and land mines

Neither K&R2 nor [33] mentions "undefined behavior", but it is a central if controversial concept in ISO C. Good summaries can be found in [2, 6, 15, 30, 40]. As described in the C "Rationale" [9], undefined behavior is a modest concept:

---

[2]There is a Clang-specific "may_alias" attribute and `memcpy` is sometimes suggested as a work-around but it introduces semantically confusing additional copying of data with hope that the optimizer may be able to do what the programmer could not do directly.

> Undefined behavior gives the implementor license not to catch certain program errors that are difficult to diagnose. It also identifies areas of possible conforming language extension: the implementor may augment the language by providing a definition of the officially undefined behavior.

This is a relatively simple, maybe deceptively simple, idea that could be interpreted, for example, as permitting a C implementation to use single machine instructions for basic arithmetic operations and let the hardware handle (or ignore) arithmetic overflow. The C standard has long declared signed integer overflow to be undefined behavior and this interpretation would permit the wrapping behavior native to most modern processors, the saturating arithmetic of some controllers, or the more widely varied behaviors of historical processors – all implemented efficiently by adopting the target processor-native semantics. This modest view of undefined behavior is not, however, the prevailing one, which is that the compiler can assume undefined behavior is impossible and can optimize on the basis of that assumption. In fact, it is currently argued that the standard interpretation allows implementations to take any action at all, not just for (say) an overflowing execution but for the entire program, if they detect a single feasible instance of undefined behavior. And there are lots of undefined behaviors.

By C18, the ISO C Standard document included a 10-page, incomplete list of undefined behaviors covering everything from type constraints to syntax errors and synchronization errors. Most C programs contain undefined behavior – certainly every operating system code base does. Perhaps more troubling, as [2] points out, this concept of undefined behavior makes C compilers unstable. A programmer may take a particular property of a C compiler for some undefined behavior to be a conforming language extension, but it may actually just be undefined behavior that has not yet been optimized. For example, gcc will generally ignore type rules on pointers - except when it does not - so that `*p = k` may work at one level of optimization, perhaps for decades, but be deleted silently when the optimizer pass recognizes a type mismatch. Kang [11] notes the "somewhat controversial practice of sophisticated C compilers reasoning backwards from instances of undefined behavior to conclude that, for example, certain code paths must be dead." can lead to "surprising non-local changes in program behavior and difficult-to-find bugs".

And by 2011, Chris Lattner, the main architect of the Clang/LLVM compilers was echoing Ritchie's warning [16]:

> To me, this is deeply dissatisfying, partially because the compiler inevitably ends up getting blamed, but also because it means that huge bodies of C code are land mines just waiting to explode. This is even worse because [...] there is

no good way to determine whether a large scale application is free of undefined behavior, and thus not susceptible to breaking in the future.

There is a proposal [35] in front of the ISO C Standards committee (WG14) to curtail undefined behavior semantics, but it is controversial.

3.1 Arithmetic Overflow

An example of how undefined behavior works in practice for arithmetic overflow was explained by Lattner [15].

> knowing that INT_MAX+1 is undefined allows optimizing X+1 > X to "true". Knowing the multiplication "cannot" overflow (because doing so would be undefined) allows optimizing X*2/2 to X.

C "ints" are fixed-size sequences of bytes interpreted as 2s complement values that map into the ring $\mathbb{Z}/2^k\mathbb{Z}$ where $(x*y)/y = x$ is not a theorem[3]. Gcc x86-64 with the optimizer on will reveal that (see the code and compilation [43]) if x is an "int" and $x = 1,000,000,000$ then calculating $(x*5)/5$ directly produces $1,000,000,000$ but also $z = x * 5 = 705032704$ and then $z/5 = 141006540$. The result depends on whether the compiler can recognize the overflows. Paradoxical results are easy to generate.

Operating system programmers in Linux discovered this issue around 2007 when they found C code of the form $if(index+length < index)\{...\}$ was being silently deleted by the compiler (since it has to be false axiomatically), causing security and logical failures [4]. Eventually, the operating system (and other projects such as the Postgres database) resorted to a compiler-specific flag to force "wrapping semantics" outside of ISO C. The same interpretation justifies "optimizing"

```
while (i++ >= i) { adjust_valve(); };
if(pressure_too_high())emergency();
```

into an infinite loop that never gets to the "if" statement.

There are many complex interactions between the "can't happen" interpretation of undefined behavior and C's rules for arithmetic and variable promotion. For example, modular arithmetic is required for *unsigned* arithmetic, but if "x" and "y" are unsigned short, and "z" is unsigned int, then the expression "z = x*y" can sometimes trigger undefined behavior. C "promotes" the two variables on the right to type "int" in order to not lose precision, but then signed integer arithmetic "can't overflow" and the compiler may sometimes assume, incorrectly, the result is less than INT_MAX [34];

3.2 What is lost

Choosing to maximise freedom for the compiler, while still specifying the language in a precise way, tends to increase

---

[3]Taking both $x \leq INT\_MAX$ and $x + 1 > x$ as axioms implies that $INT\_MAX > INT\_MAX$.

the burden of complexity shared by implementors and users. Consider these optimization-related issues.

3.2.1 Pointer casts. According to the text of the C18 ISO C standard, pointers of most types can be freely cast to other pointer types (section 6.3.2.3 paragraph 7).

> A pointer to an object type may be converted to a pointer to a different object type. If the resulting pointer is not correctly aligned 69) for the referenced type, the behavior is undefined.

However, complex rules govern what accesses are permitted via such pointers. The pointer is correctly aligned in this code:

```
float *f = malloc(sizeof(float));
*f = 3.14; //1
int *a =  (int *)f
*a = 4; //2
```

The language in section 6.5 paragraph 6 then implies that the "effective type" of an allocated object can be changed by writing to it:

> If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value.

So have we converted the object pointed to by f to an int in statement 2? Section 6.5 paragraph 7 says " *An object shall have its stored value accessed only by an lvalue expression that has one of the following types*", which are limited to compatible types and character types. We can convert f to be a pointer to an int, which is even correctly aligned, but "access" includes both reads and writes. The assignment of statement 2 is (currently) compiled as written by both gcc and Clang, although the similar floating-point example above omits the assignment to the aliasing pointer. The rules here were substantially revised for C99, but are still not considered adequate (e.g. even by many in WG14's Memory Object Model study group). In C89 this same section allows access only by lvalues that have compatible *declared* types, which appears to prevent any access at all to allocated objects coming from malloc, so statement 1 would be undefined behavior in C89 – apparently an error in the specification. Derek Jones [10] points out other changes in the C99 standard were required to allow memcpy to be written in C, following mistakes in C89 that prevented this.

Related issues include the behaviour of freed pointers [22], comparison of 'one-past-the-end' pointers [23], and semantics of integer-to-pointer casts [11]. The latter proposes a 'quasi-concrete' semantics, where casting a pointer to an integer limits the permitted optimizations. This restraint is relatively rare – it is unclear whether its compromise will

be acceptable to ISO – yet still misses some cases where addresses may be validly known to the wider program.

3.2.2 Temporally unbounded UB.
The WG14 Memory Object Model study group was started in order to come up with a proposal for memory and pointer semantics. Their working proposal [5] is quoted twice above, but also explains:

> For evaluation-order and concurrency nondeterminism, one would normally say that if there exists any execution that flags UB, then the program as a whole has UB (for the moment ignoring UB that occurs only on some paths following I/O input, which is another important question that the current ISO text does not address). This view of UB seems to be unfortunate but inescapable. If one looks just at a single execution, then (at least between input points) we cannot temporally bound the effects of an UB, because compilers can and do re-order code w.r.t. the C abstract machine's sequencing of computation. In other words, UB may be flagged at some specific point in an abstract-machine trace, but its consequences on the observed implementation behaviour might happen much earlier (in practice, perhaps not very much earlier, but we do not have any good way of bounding how much). But then if one execution might have UB, and hence exhibit (in an implementation) arbitrary observable behaviour, then anything the standard might say about any other execution is irrelevant, because it can always be masked by that arbitrary observable behaviour.

Perhaps the unifying theme of both of these topics can be found in Dennis Ritchie's comment on noalias cited above:*"I don't know any acceptable way of changing the language specification to express the possibility of this kind of optimization"*. C is stubbornly low-level and changing the language specification to permit these types of optimizations is hard, or maybe impossible. Current proposals introduce highly complex rules, which despite their complexity are known to be inadequate for certain systems programming idioms. If accepted by ISO, they are likely to be misunderstood by both practitioners and implementors, perpetuating rather than solving the problem. As one possible solution, Torvalds [39] and Ertl [3] both propose relatively concrete, operational alternatives – where compilers map source operations to well-defined instruction sequences, in either a virtual or real machine, from which compiler optimisations may not observably stray.

4 What is gained
The second part of Ritchie's comment cited above is *"and I don't know how much performance improvement is likely to result."* It is difficult to find any documentation of significant performance advantages of any kind of undefined behavior optimization [2]. The standard TBAA lifting example can be better and more generally optimized by hand without much effort – without needing a type mismatch. Wang *et al* ([40], Sec. 3.3) could not find a case where a UB optimization could not be matched with simple coding changes. Other optimizations are also mostly justified by small differences in SPEC benchmarks or references to proprietary data sets.

Lee [17] provides an example of the claimed advantage of undefined behavior for overflow.

```
for(int i=0 ; i <= N; i++)a[i] = x+1;
```

If i and N are both 32bit and the target machine is x86-64, it is sometimes assumed that permitting overflow requires a sign extend of i on each iteration.

```
.L3:    movsx   rcx, eax #64 bit sign extend i
        add     eax, 1
        mov     DWORD PTR [rsi+rcx*4], edx
        cmp     edi, eax
        jge     .L3
```

Assuming overflow is impossible allows omitting the sign extend. Sign extension is a fast operation so if the loop is non-trivial, the cost will be lost in the noise. In any case, i can only overflow when N == INT_MAX. The code then executes an infinite loop, writing x+1 from $a[INT\_MIN]$ to $a[INT\_MAX]$. Gcc will omit the sign extend all the same, if the programmer replaces <= with <. In sum, the "slowdown" is caused by a single case that is nearly certainly an error and easily avoided. As with many other similar cases, this example of ISO C optimizations turns out to to depend on the assumption that C programmers will not profile and optimize their own code.

Even for aliasing-based optimizations, it is not necessary for alias analysis to depend on undefined behavior. Alias detection in the abstract is not Turing computable [29] and C pointers make approximate alias detection difficult [14], but there are effective algorithms that can detect most aliasing [7] and it is a design choice to make aliasing optimization rely on assumptions about program code that are not validated. ISO C has chosen to reduce the burden on the compilers at the expense of semantic clarity.

The CompCert compiler is aimed at control systems that have many of the same properties as operating systems, does not do any undefined behavior based optimization[4] (and does not optimize extensively) [19] and has a deterministic semantics [18]:

> The semantics is deterministic and makes precise a number of behaviors left unspecified or undefined in the ISO C standard [...]
> CompCert generates code that is more than twice as fast as that generated by gcc without optimizations, and competitive with gcc at optimization

---

[4]Except for assuming objects do not overlap in memory.

levels 1 and 2. On average, CompCert code is only 7% slower than gcc -O1 and 12% slower than gcc -O2.

Finally, there is an experiment done at RedHat by Vladimir Makarov [21]:

> I did an experiment by switching on only a fast and simple RA and combiner in gcc. There are no options to do this, I needed to modify gcc. [..] Compared to hundreds of optimizations in gcc-9.0 with -O2, these two optimizations achieve almost 80% performance on an Intel i7-9700K machine under Fedora Core 29 for real-world programs through SpecCPU, one of the most credible compiler benchmarks.

Makarov then tested a 20 year old version of gcc on Spec benchmarks versus a contemporary version of the compiler to show a 16% improvement with all optimizations enabled — over a period where most UB based optimizations went into the compilers.

A small performance improvement will generally not justify a decrease in code stability for operating systems or avionics controllers, but the answer may be different for "at-scale" data center applications or large numerical simulations.

A Acknowledgments

References

[1] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2012. Understanding Integer Overflow in C/C++. *Proceedings - International Conference on Software Engineering* 25 (07 2012). https://doi.org/10.1109/ICSE.2012.6227142

[2] M. Anton Ertl. 2015. What every compiler writer should know about programmers. In *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'15)*, Jens Knoop and M. Anton Ertl (Eds.). 112–133. http://www.complang.tuwien.ac.at/kps2015/proceedings/KPS_2015_submission_29.pdf

[3] M. Anton Ertl. 2017. The Intended Meaning of *Undefined Behaviour* in C Programs. In *19. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'17)*, Wolfram Amme and Thomas Heinze (Eds.). 20–28. http://www.complang.tuwien.ac.at/papers/ertl17kps.pdf

[4] Felix-gcc. 2007. Bug 30475 - assert(int+100 > int) optimized away. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475

[5] Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, and Martin Uecker. 2021. A Provenance-aware Memory Object Model for C. Draft Technical Specification N2577. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2577.pdf

[6] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 336–345. https://doi.org/10.1145/2737924.2737979

[7] M. Hind and Anthony Pioli. 2000. Which pointer analysis should I use?. In *ISSTA '00*.

[8] ISO PL22.11 - SC22/WG14 . 2018. *Programming language: C: ISO/IEC 9899:2018 (C18)*. Number ISO/IEC 9899:2018).

[9] INCITS J11 and SC22 WG14. 2003. Rationale for International Standard. Programming Languages. C Revision 5.10. http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf

[10] Derek Jones. 2017. How indeterminate is an indeterminate value. http://shape-of-code.coding-guidelines.com/2017/06/18/how-indeterminate-is-an-indeterminate-value/

[11] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A Formal C Memory Model Supporting Integer-Pointer Casts. *SIGPLAN Not.* 50, 6 (June 2015), 326–335. https://doi.org/10.1145/2813885.2738005

[12] Brian W. Kernighan. [n.d.]. Why Pascal is Not My Favorite Programming Language. http://www.lysator.liu.se/c/bwk-on-pascal.html

[13] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language* (2nd ed.). Prentice Hall Professional Technical Reference.

[14] W. Landi and B. Ryder. 1992. A safe approximate algorithm for interprocedural aliasing. In *PLDI '92*.

[15] Chris Lattner. 2011. What every C programmer should know. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

[16] Chris Lattner. 2011. What Every C Programmer Should Know About Undefined Behavior 2/3. https://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html

[17] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. *SIGPLAN Not.* 52, 6 (June 2017), 633–647. https://doi.org/10.1145/3140587.3062343

[18] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://xavierleroy.org/publi/compcert-CACM.pdf

[19] Xavier Leroy. 2021. Personal Communication.

[20] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA. 26 pages. https://hal.inria.fr/hal-00703441

[21] Vladimir Makarov. 2020. MIR: A lightweight JIT compiler project. https://developers.redhat.com/blog/2020/01/20/mir-a-lightweight-jit-compiler-project

[22] Paul E. McKenney, Maged Michael, Jens Mauer, Peter Sewell, Martin Uecker, Hans Boehm, Hubert Tong, and Niall Douglas. 2019. Pointer lifetime-end zap. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1726r0.pdf

[23] Joseph Myers. 2014. "Bug 61502: comparison on "one-past" pointer gives wrong result, comment 1". https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61502

[24] Thi Viet Nga Nguyen and François Irigoin. 2003. Alias verification for Fortran code optimization. *J. UCS* 9, 3 (2003), 270.

[25] Vegard Nossum. 2016. "Subject [PATCH] firmware: declare __start,end_builtin_fw as pointers". https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61502

[26] X3J11 Technical Committee on the C Programming Language. 1989. ANSI X3.159, 1989 Edition, 1989 - INFORMATION SYSTEMS - PROGRAMMING LANGUAGE - C.

[27] Clang Project. 2021. Clang 13 Documentation. https://clang.llvm.org/docs/UsersManual.html

[28] GNU Project. 2021. GCC Documentation. https://gcc.gnu.org/onlinedocs/

[29] Ganesan Ramalingam. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1467–1471.

[30] John Regehr. 2010. https://blog.regehr.org/archives/213

[31] John Regehr. 2016. The Strict Aliasing Situation is Pretty Bad. https://blog.regehr.org/archives/1307

[32] Dennis Ritchie. 1988. noalias comments to X3J11. (March 1988). https://groups.google.com/g/comp.lang.c/c/K0Cz2s9il3E/m/YDyo_xaRG5kJ

[33] DM Ritchie, SC Johnson, ME Lesk, and BW Kernighan. 1978. The C programming language, Bell Systems Tech. *J* 57, 6 (1978), 1991–2020.

[34] Eskil Steenberg. 2021. "Compiler Explorer UShort promotion UB". https://godbolt.org/z/7q9dPzEfM

[35] Eskil Steenberg. 2021. Redefining Undefined Behavior N2769. (21 6 2021). http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2769.pdf

[36] Linus Torvalds. [n.d.]. Reloc-Hide in Linux Kernel. https://github.com/torvalds/linux/blob/35e43538af8fd2cb39d58caca1134a87db173f75/include/linux/compiler-gcc.h

[37] Linus Torvalds. 2009. Re Gcc inlining heuristics. https://www.mail-archive.com/linux-btrfs@vger.kernel.org/msg01647.html

[38] Linus Torvalds. 2018. Re: [GIT PULL] Device properties framework update for v4.18-rc1. https://lkml.org/lkml/2018/6/5/769

[39] Linus Torvalds. 2018. Re: LKMM litmus test for Roman Penyaev's rcu-rr. https://lkml.org/lkml/2018/6/7/761

[40] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined Behavior: What Happened to My Code?. In *Proceedings of the Asia-Pacific Workshop on Systems* (Seoul, Republic of Korea) *(APSYS '12)*. Association for Computing Machinery, New York, NY, USA, Article 9, 7 pages. https://doi.org/10.1145/2349896.2349905

[41] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2015. A Differential Approach to Undefined Behavior Detection. *ACM Trans. Comput. Syst.* 33, 1, Article 1 (March 2015), 29 pages. https://doi.org/10.1145/2699678

[42] William A Wulf. 1972. Systems for systems implementors: some experiences from Bliss. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part II*. 943–948.

[43] Victor Yodaiken. 2021. Compiler Explorer ISO C Division. https://godbolt.org/z/zWh9c5e84

[44] Victor Yodaiken. 2021. Example of Clang and type based alias. https://godbolt.org/z/nq19n8dhE