Catch-23: The New C Standard Sets the World on Fire

TERENCE KELLY
WITH SPECIAL GUEST BORER YEKAI PAN

A new major revision of the C language standard, C23, is due out this year. We'll tour the highs and lows of the latest draft[9] and then drill down on the mother of all breaking changes. Sidebars celebrate C idioms and undefined behavior with code and song, respectively.

THE GOOD NEWS

Like the previous major revision, C11,[7] the latest standard introduces several useful features. The most important, if not the most exciting, make it easier to write safe, correct, and secure code. For example, the new `<stdckdint.h>` header standardizes *checked integer arithmetic*:

```
int i =...; unsigned long ul =...; signed char sc =...;
bool surprise = ckd_add(&i, ul, sc);
```

The type-generic macro `ckd_add()` computes the sum of `ul` and `sc` "as if both operands were represented in a signed integer type with infinite range." If the mathematically correct sum fits into a signed int, it is stored in `i` and the macro returns `false`, indicating "no surprise"; otherwise, `i` ends up with the sum wrapped in a well-defined way and the macro returns `true`. Similar macros handle multiplication and subtraction. The `ckd_*` macros steer a refreshingly sane path around arithmetic pitfalls including C's "usual arithmetic conversions."

C23 also adds new features to protect secrets from prying eyes and programmers from themselves. The new `memset_explicit()` function is for erasing sensitive in-memory data; unlike ordinary `memset`, it is intended to prevent optimizations from eliding the erasure. Good old `calloc(size_t n, size_t s)` still allocates a zero'd array of `n` objects of size `s`, but C23 requires that it return a null pointer if `n*s` would overflow.

In addition to these new correctness and safety aids, C23 provides many new conveniences: Constants `true`, `false`, and `nullptr` are now language keywords; mercifully, they mean what you expect. The new `typeof` feature makes it easier to harmonize variable declarations. The preprocessor can now `#embed` arbitrary binary data in source files. Zero-initializing stack-allocated structures and variable-length arrays is a snap with the new standard "`={}`" syntax. C23 understands binary literals and permits apostrophe as a digit separator, so you can declare `int j = 0b10'01'10`, and the `printf` family supports a new conversion specifier for printing unsigned types as binary ("01010101"). The right solution to the classic job interview problem "Count the 1 bits in a given `int`" is now `stdc_count_ones()`.

Sadly, good news isn't the only news about C23. The new standard's nonfeatures, misfeatures, and defeatures are sufficiently numerous and severe that programmers should not "upgrade" without carefully weighing risks against benefits. Older standards such as C99 and C11 weren't perfect, but detailed analysis will sometimes conclude that they are preferable to C23.

After reviewing C23's problems, we'll discuss strategies for peaceful coexistence with existing code and hazard mitigation in new code.

> **"C inventor Dennis Ritchie pointed to several flaws in [ANSI C] ... which he said is a licence for the compiler to undertake agressive opimi-sations that are completely legal by the committee's rules, but make hash of apparently safe programs; the confused attempt to improve optimisation ... spoils the language."**
>
> *—Dennis Ritchie on the first C standard* [4, 27]

UNFILLED POTHOLES AND FESTERING SORES

Laws should be freely available, intelligible, and agreeable to the governed, and they should keep pace with changing times. C23 lacks these virtues.

Standard C hides behind a paywall: The official standard currently costs more than $200, so most coders make do with unofficial drafts.[1] The standard routinely confuses its own authors, and crucial parts mystify even experienced and well-educated programmers[27]; baffled silence is not consent. Developers who manage to figure out what the standard actually means are frequently appalled.[25,27] Standard C advances slowly (e.g., 30 years and five revisions to define zero equal to zero[26]) and sometimes not at all.

Progress means draining swamps and fencing off tar pits, but C23 actually *expands* one of C's most notorious traps for the unwary. All C standards from C89 onward have permitted compilers to delete code paths containing undefined operations—which compilers merrily do, much to the surprise and outrage of coders.[16] C23 introduces a *new* mechanism for astonishing elision: By marking a code path with the new `unreachable` annotation,[12] the programmer assures the compiler that control will never reach it and thereby explicitly invites the compiler to elide the marked path. C23 furthermore gives the compiler license to use an `unreachable` annotation on one code path to justify removing, without notice or warning, an entirely *different* code path that is *not* marked `unreachable`: see the discussion of `puts()` in Example 1 on page 316 of N3054.[9]

Major disappointments of inaction involve the pillar of C programming: pointers. Comparing pointers to different objects (different arrays or dynamically allocated blocks of memory) is still undefined behavior, which is a polite way of saying that the standard permits the compiler to run mad and the machine to catch fire at run time.[16] The standard's pointer comparison restrictions, rooted in forgotten ancient hardware architectures, have surprising consequences. The seemingly innocent sequence `a=malloc(...)` then `b=malloc(...)` then `if (a<b)...` is a recipe for conflagration, and it's impossible to implement the standard `memmove()` function efficiently in standard C.[16] Furthermore, after much discussion, the arcane and poorly motivated "pointer zap" rules[21] remain in effect: Pointers to `free`'d memory are akin to uninitialized pointers, so `free(p)` followed by `if (p==q)...` is an instrument of arson. Things need not be so.

C23 fails to correct misguidance dating to the earliest version of the standard. Its example implementation of `rand()` is still the same primitive linear congruential generator returning *16-bit* integers—a design that was ripe for taxidermy at the turn of the century. XORshift random number generators, invented 20 years ago, would make a better example: They are simple and fast, accommodate 32-, 64-, and 128-bit machine words, and produce superior random sequences.[20]

Developers should also note that C23 has drifted further from C++ than the earlier C standards. The notion that C is (mostly) a subset of C++ is further from reality than ever before.[10]

Sadly, missed opportunities and incompatibilities with C++ aren't the worst aspects of the new standard. C23 transforms decades of perfectly legitimate programs into Molotov cocktails.

INCENDIARY `realloc()`

The `realloc` function, standard since C89, resizes a memory allocation. C23 senselessly outlaws a useful `realloc` feature that was very deliberately designed and blessed by C89 through C11, rendering C23 `realloc` far less versatile and stuffing tinder into myriad programs written to earlier standards. To understand the folly of the recent ban, we must review the full-featured `realloc` of yesteryear and the elegant idiom to which it is perfectly suited.

C89 defined `realloc` to include `malloc` and `free` as special cases:

```
void *realloc(void *ptr, size_t size);
```

> "The `realloc` function changes the size of the object pointed to by `ptr` to the size specified by `size`.... If `ptr` is a null pointer, the `realloc` function behaves like the `malloc` function for the specified size...If the space cannot be allocated, the object pointed to by `ptr` is unchanged. If `size` is zero and `ptr` is not a null pointer, the object it points to is freed."
>
> *— C89,[2] repeated verbatim in Plauger[22]*

Plenty of real-world code exploits the versatility of realloc. Examples include dozens of executables on the search $PATH of Linux machines:

```
$ echo foo | ltrace grep bar |& grep realloc
realloc(0, 128)      = 0x55a17f5596f0
```

The C89 and C99 standards committees strongly recommended that allocation interfaces malloc, calloc, and realloc return a null pointer in response to zero-byte requests.[3,6] This implies that realloc(p,0) should unconditionally free(p) and return NULL: No new allocation happens in this case, so there's no possibility of an allocation failure. For brevity, let "*zero-null*" denote allocator implementations that comply with the C89/C99 guidance.

The Swiss-Army-knife aspect of realloc is daunting at first, but this interface rewards patient study. Soon you realize that zero-null realloc was thoughtfully designed to enable elegant dynamic arrays that do exactly the right thing under all circumstances, obviating the need for clunky and error-prone code to handle grow-*from*-zero and shrink-*to*-zero as special cases.

Figure 1 illustrates idiomatic realloc via a simple stack that grows with every push() and shrinks with every pop(). Pointer S and counter N (lines 1 and 2) represent the stack: S points to an array of N strictly positive ints. Because they are statically allocated, initially the pointer is NULL and the counter is zero, indicating an empty stack. Function resize (lines 4–10) resizes the stack to a given new capacity, checking for arithmetic overflow (line 6) before calling realloc and checking the return value for memory

FIGURE 1: **CONTINUOUSLY RIGHTSIZING A STACK WITH ZERO-NULL** `realloc`

```
1  static int   * S;  /* array of strictly positive ints */
2  static size_t N;  /* number of ints on stack */

4  static int resize(const size_t nu) {
5    int *t;
6    if (nu > SIZE_MAX / sizeof *S)          return TOOBIG;
7    t = realloc(S, nu * sizeof *S);
8    if (nu && !t)  /* ask >0, get none */   return ALLOCFAIL;
9    S = t;
10   N = nu;  /* NB: side effect */          return 0;   }

12 int push(const int val) {                  /* all error */
13   int r;                                   /* codes are */
14   if (0 >= val)             return BADARG; /* negative; */
15   if ((r = resize(N+1)))    return r;      /* see       */
16   S[N-1] = val;             return 0;   }  /* stack.h   */

18 int pop(void) {
19   int r, topval;
20   if (!N)                   return EMPTY;
21   topval = S[N-1];
22   if ((r = resize(N-1)))    return r;
23   else                      return topval;   }
```

exhaustion (line 8). Allocation failure is inferred when a *nonzero* new size is requested but `NULL` is returned; zero-null `realloc` also returns `NULL` when the second argument is zero, but this does not indicate an allocation failure because no allocation was attempted. (Checking `errno` doesn't enable portable code to detect allocation failure because the C standards don't say how out-of-memory affects `errno`.) Thanks to zero-null `realloc`'s versatility, the `resize` function need not consider whether the stack is growing from zero or shrinking to zero or re-sizing in some other way; everything Just Works regardless.

The code of figure 1 follows a few simple rules implicit in the semantics of zero-null `realloc`. Functions `push` and `pop` (lines 12–23) access the stack only via subscripts on `S`, because `realloc` may move the array to a different location in memory. They never dereference `S` when `N` is zero. The `resize` function resists the temptation of reckless `S = realloc(S,...)`, which destroys the entry point into the array when allocation fails, thereby leaking memory and losing data.

I've been seeing code resembling figure 1 for 30 years, starting with the work of an older schoolmate who had bothered to read the fine manual; the clarity and simplicity of his code left a deep impression. In the decades since then I have repeatedly found idiomatic `realloc` in serious production code, usually while scanning for `p = realloc(p,...)` bugs.

Imagine, then, my dismay when I learned that C23 declares `realloc(ptr,0)` to be undefined behavior, thereby pulling the rug out from under a widespread and exemplary pattern deliberately condoned by C89 through C11. So much for *stare decisis*. Compile idiomatic `realloc` code as C23 and the compiler might maul the source in most astonishing ways and your machine could ignite at runtime.[16] To make matters much worse, recompilation is not a prerequisite for conflagration: Merely re-*linking* existing compiled binaries with a new or "upgraded" standard library sets the stage for disaster. If your standard library is implemented as a dynamically linked shared library (e.g., `libc.so`), running a binary executable from yesteryear will load the latest library at run time, so have a fire extinguisher on hand when you upgrade that shared library to C23. Every program that uses `realloc` as `free` in the manner intended by three generations of standards is an inferno waiting to happen, and the legions of programmers accustomed to classic versatile `realloc` need re-education.

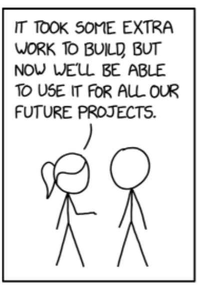
The immediate explanation for this disastrous change is remarkably unconvincing and apparently took scant notice of decades of sound idiomatic usage: Essentially, "implementations of `realloc(p,0)` differ, so let's scrap the lot,"[23] which inverts a boldface tenet of C standardization: "**Existing code is important, existing implementations are not.**"[3,6] The full sad history reveals a much larger and more troubling problem that bodes ill for the future of C.

THE (COM)PROMISED LAND

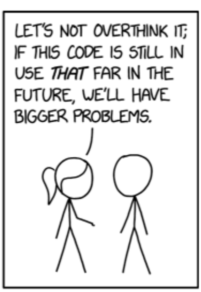
Standards are supposed to lead the way to a better world by making portable code possible. Genuine standardization inevitably requires herding cats—permitting diverse compiler and library implementations to flourish while enforcing sensible behavior. The chronicle of the `realloc` affair shows that C standardization doesn't work that way nowadays.

As C89 was taking shape, the notion of a "zero-length object" was making the rounds: Proponents argued that a non-null pointer to such an object should be returned for zero-byte allocation requests.

Why are such requests made? Often because of arithmetic bugs. And what is a non-null pointer from `malloc(0)` good for? Absolutely nothing, except shooting yourself in the foot.



HOW TO ENSURE YOUR CODE IS NEVER REUSED



HOW TO ENSURE YOUR CODE LIVES FOREVER

https://xkcd.com/2730/

It is illegal to dereference such a pointer or even compare it to any other non-null pointer (recall that pointer comparisons are combustible if they involve different objects). Scour the annals of computing and you'll find few things more perfectly useless than a zero-length object and few things more hazardous than a pointer thereto. Not surprisingly, analogs are rare in the world beyond computing: Try depositing a check in the amount of $0 into your bank account.

Both C89 and C99 wisely "decided not to accept the idea of zero-length objects," but foolishly failed to ban it. As we saw earlier, they strongly recommended zero-null allocation, but they also reluctantly allowed `malloc(0)` to return non-null as an amnesty for wayward implementations that already did so.[3,6] Which means that `realloc(p,0)` might need to allocate a new zero-length object. And this allocation might fail (just as an attempt to deposit cash into a bank account and simultaneously withdraw $0 might fail—say it to yourself slowly, savoring the absurdity). By the time C17 was in the works, implementations that attempted to allocate a zero-length object for `realloc(p,0)` disagreed about whether `free(p)` should happen if the allocation fails. So C17 made this behavior implementation-defined and declared `realloc`-as-`free` obsolescent,[8] setting the stage for C23's outright ban.

To summarize, this downward spiral began with a concept worthy of Monty Python that snowballed and metastasized, thanks to a feckless compromise. The C23 `realloc` mess is just the tip of the iceberg. The root problem is the failure of a standard to standardize.

Looking forward, marijuana legalization will surely beget notions such as fractional-, imaginary-, and negative-length objects, each with as much potential for mayhem as zero-length objects. Let us hope that future standards committees will work up the courage to do more than survey the status quo, sprinkle most of it with holy water, and consign to flame whatever actually needs to be standardized.

MUDDLING THROUGH

How should you respond to C23? Understand its implications for both your existing code and for code yet unwritten. Compile old code as C23 only for good reason and only after verifying that it doesn't run afoul of any constriction in the new standard. If you need new C23 features, consider quarantining C23 code in separate translation units; fortunately, object-code files compiled from different source dialects can be linked together. Beware that changes to the standard library could impose unwelcome new semantics—or abolish required old semantics, as with `realloc`—and that such changes may impose themselves on old code *without* recompilation when dynamically linked libraries are upgraded.

If you're the sort of person who thinks independently and insists that the tools of the trade be intelligible and sensible, you're in the majority. Work with your colleagues to lobby your compiler and library vendors and the standards committee (wg14@soasis.org) if things aren't to your liking.

> **"Standards are not some kind of holy book that has to be revered. Standards too need to be questioned."**
>
> *— Linus Torvalds on C standards*[25]

DRILLING DEEPER

To write new code, you must track current language standards; to maintain old code, you must understand earlier ones. Kernighan and Ritchie[18] provide the classic account of C89[2]; Plauger documents its standard library.[22] Harbison and Steele[13] cover C99.[5] Klemens[19] explains useful features introduced in C11.[7] Hatton details precautions for safety-critical C coding.[14]

Bits

Download the example code at https://queue.acm.org/downloads/2023/Drill_Bits_09_example_code.tar.gz. You get the stack of figure 1 and simple wrapper code that transforms *any* standards-compliant memory allocator into a zero-null allocaor.

Drills

1. Figure 1's stack sacrifices speed for clarity and brevity. Implement a more efficient design that separately tracks capacity and item count, resizing capacity by 2x as appropriate.[17]
2. If your `malloc(0)` returns non-null, how many calls does it take to exhaust memory? How many $0 withdrawals does it take to bankrupt your bank?
3. Use the new C23 `#embed` feature to implement literate executables.[15]
4. Search for the `p = realloc(p,...)` bug in real code and textbooks (e.g., page 253 of Klemens[19]). See also page 101 of the C89 Rationale[3] and page 160 of the C99 Rationale.[6]
5. If you think idiomatic C is cryptic, recall the old joke

about the Perl mafioso: He makes you an offer you can't understand. List the best idioms and worst abuses of your favorite languages.

6. Check the `#define` of `INT_MIN` in `<limits.h>`. If you see something like `(-INT_MAX - 1)`, why isn't it more straightforward? See page 46 of Gustedt.[11]
7. C17[8] purports to be a bug-fix revision of C11. Does the word "`toto`" on page 1 indicate (a) the editor's musical tastes; (b) that nobody bothered to spell-check the document; (c) that we're not in Kansas anymore; or (d) none of the above?
8. Programmer Yossarian's application requires the new C23 `memset_explicit` function but also requires `realoc(p,0)` to be well defined. If both functions live in `libc.so`, is Yossarian caught in a Catch-23? What should he do?
9. Following Shiffman,[24] write a Socratic dialogue in which C inventor Dennis Ritchie interrogates the C standards committee. See Yodaiken[27] for talking points.

Idioms and Fluency

I decry C23's ban on the elegant idiomatic use of a classic memory allocation function. Why should you care? What's so important about idioms?

Fluent, idiomatic code expresses the programmer's intentions more accurately, more clearly, and often more succinctly than rookie code. C's idioms are not excessively numerous or abstruse, but to master them

```
if (n != 0)
    b = 1;
else
    b = 0;


b = n ? 1 : 0;


b = n != 0;


b = !!n;
```

you must climb a learning curve. For example, the snippets above show how most C programmers learn to collapse a numeric variable to a Boolean. The "bang-bang" idiom at the bottom isn't best in every situation, but it's often handy and you must recognize it to read expert code.

Idiomatic expression proves its practicality when cluttery alternatives would complicate an inherently simple chore. For example, imagine a hotel that charges extra for pets: The first dog costs $10 and each additional pooch $5 more; likewise for other animals but with different constants. Idiomatic code is natural, easy to write, compact, and obvious to fluent readers:

```
petFee = !!ndogs * (10 + (ndogs-1) * 5)  // risk: rugs
       + !!ncats * ( 7 + (ncats-1) * 3)  // " furniture
       + !!nfish * (47 + (nfish-1) * 1); // " floods
```

Bang-bang, like most C idioms, is based not on esoteric knowledge but rather on a thorough understanding of fundamentals. Beginners who overlook the bang-bang option know about logical NOT, but perhaps they assume that double negation can't be useful. Fluent programmers, however, appreciate the peculiar nuances of the "!" operator. Mastering double-bang is largely a matter of fully understanding single-bang.

Kernighan and Pike discuss programming idioms at length.[17] Klemens describes cool idioms enabled by C11 features.[19] Yodaiken explains how aspects of the C standards intended to enable performance optimizations undermine systems programming idioms.[27]

Exercise: Amend the petFee formula above to add premiums for *combinations* of species that don't play nice together: $30 for any numbers of dogs and cats, because noise; and $20 for any numbers of cats and fish, because splashes. Hint: What happens when you multiply Booleans?

Undefined Behavior Acid Trip

This parody of Jefferson Airplane's classic song "White Rabbit" is about programming psychedelia—undefined behavior in C. The title refers to the empty assembly-code file you get when the compiler elides code paths with UB.[16] Helgrind is a tool in the Valgrind suite. Chris Lattner created the Clang compiler.

white.s

One flag makes it faster
and one flag makes it small
and the deprecated `-Wchkp`
doesn't do anything at all.
Go ask Lattner
if we should use `-Wall`.

And if you go comparing pointers
across segments you're going to fall.
That's how a hookah-smoking working group
has standardized it all.
Go ask Lattner
did they make the right call?

When your loops and expressions
get up from where you said they go
and Clang just had some kind of warning
and Valgrind is moving slow,
go ask Lattner;
I hope he'll know.

When the logic of `-O3`
is calling your code dead
and the `main()` task is writing backwards
while the workers race ahead,
remember what the Helgrind said:
Lock your thread. Lock your thread.

Acknowledgments

Jon Bentley, Hans Boehm, John Dilley, Kevin O'Malley, and Charlotte Zhuang reviewed drafts and provided helpful feedback. Dilley and O'Malley scrutinized the example code and recommended useful improvements. Dhruva Chakrabarti and Pramod Joisha fielded technical questions. We thank C23 standards committee members Aaron Ballman, Robert Seacord, and JeanHeyd Meneide for exchanges of correspondence.

References

1. C Standards Committee (Working Group 14). Documents; https://www.open-std. org/jtc1/sc22/wg14/www/wg14_document_log.htm.
2. C89 Standard; https://web.archive.org/web/20161223125339/http:// flash-gordon.me.uk/ansi.c.txt.
3. C89 Rationale, ANSI X3J11/88–151, November 1988. Available via https://en. wikipedia.org/wiki/ANSI_C.
4. *Computer Business Review* staff. 1988. Proposed ANSI C language standard draws criticism as comment period ends; https://techmonitor.ai/technology/proposed_ansi_c_ language_standard_draws_criticism_as_comment_period_ends.
5. C99 Standard (draft n1256). 2007. https://www.open-std.org/jtc1/sc22/ WG14/www/docs/n1256.pdf.
6. C99 Rationale. 2003. https://www.open-std.org/jtc1/sc22/wg14/www/ C99RationaleV5.10.pdf.
7. C11 Standard (draft n1570). 2011. http://www.open-std.org/jtc1/sc22/wg14/www/ docs/n1570.pdf.
8. C17 Standard (draft n2176). https://web.archive.org/

web/20181230041359/ http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_ proposed_fdis.pdf.
9. C23 Standard (draft n3054). 2022. https://www.open-std.org/jtc1/sc22/wg14/ www/docs/n3054.pdf.
10. Ballman, A. 2022. WG14 document n3065: C xor C++ programming; https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3065.pdf. [Also available at Reference 1.]
11. Gustedt, J. 2019. *Modern C*, second edition. Manning; https://gustedt.gitlabpages.inria.fr/modern-c/.
12. Gustedt, J. 2021. WG14 document n2826v2. Add annotations for unreachable control flow; https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2826.pdf. [Also available at Reference 1.]
13. Harbison, S. P., Steele III, G. L. 2002. *C: A Reference Manual,* fifth edition. Prentice Hall.
14. Hatton, L. 1995. *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*. McGraw-Hill.
15. Kelly, T. 2022. Literate executables. *acmqueue* 20(5); https://queue.acm.org/detail.cfm?id=3570938.
16. Kelly, T., Gu, W., Maksimovski, V. 2021. Schrödinger's code: undefined behavior in theory and practice. *acmqueue* 19(2); https://queue. acm.org/detail.cfm?id=3468263.
17. Kernighan, B., Pike, R. 1999. *The Practice of Programming*. Addison-Wesley.
18. Kernighan, B. W., Ritchie, D. M. 1988. *The C Programming Language,* second edition. Prentice Hall.
19. Klemens, B. 2014. *21st Century C,* second edition. O'Reilly Media.
20. Marsaglia, G. 2003. Xorshift RNGs. *Journal of Statistical Software* 8(14); https://www.jstatsoft.org/

index.php/jss/article/view/v008i14/916.
21. McKenney, P. E., Michael, M., Mauer, J., Sewell, P., Uecker, M., Boehm, H., Tong, H., Douglas, N., Rodgers, T., Deacon, W., Wong, M. 2019. WG14 document n2443: Lifetime-end pointer zap; https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2443.pdf. [Also available at Reference 1.]
22. Plauger, P. J. 1992. *The Standard C Library*. Prentice Hall.
23. Seacord, R. C. 2019. WG14 document n2464: Zero-size reallocations are undefined behavior; https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2464.pdf. [Also available at Reference 1.]
24. Shiffman, M. 2022. A man of all reasons. *Harper's Magazine* (April), 15–16; https://harpers.org/archive/2022/04/steven-pinker-meets-socrates/.
25. Torvalds, L. 2018. Linux kernel mailing list posting; https://lkml.org/ lkml/2018/6/5/769.
26. C FP Group. 2021. WG14 document n2670: Zeros compare equal; https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2670.pdf. [Also available at Reference 1.]
27. Yodaiken, V. 2021. How ISO C became unusable for operating systems development. 11th Workshop on Programming Languages and Operating Systems (PLOS '21). https://doi.org/10.1145/3477113.3487274.

Terence Kelly (tpkelly@acm.org) *and* Yekai Pan *enjoy surveying the status quo, sprinkling most of it with holy water, and consigning to flame the parts they don't like.*

CONTENTS