Schrödinger's Code

UNDEFINED BEHAVIOR IN THEORY AND PRACTICE

TERENCE KELLY WITH SPECIAL GUEST BORERS

WEIWEI GU AND VLADIMIR MAKSIMOVSKI

SANITY VS. SPEED

Undefined behavior ranks among the most baffling and perilous aspects of popular programming languages. This installment of Drill Bits clears up widespread misconceptions and presents practical techniques to banish undefined behavior from your own code and pinpoint meaningless operations in any software—techniques that reveal alarming faults in software supporting business-critical applications at Fortune 500 companies.

Early in the history of programming languages, two schools of thought diverged. Quicksort inventor C.A.R. Hoare summarized one philosophy in his Turing Award lecture:⁷ The behavior of every syntactically correct program should be completely predictable from its source code. For the sake of safety, security, and programmer sanity, it must be impossible for a program to “run wild.” Ensuring well-defined behavior imposes runtime overheads (e.g., array bounds checks), but predictability justifies the cost. Today, “safe” languages such as Java embody Hoare’s advice.

A different philosophy reigns in domains that demand efficiency and speed (e.g., infrastructure software). Systems programming languages such as C and C++ sacrifice safety and comprehensive semantics for

performance. These languages, despite being meticulously standardized, do *not* define the behavior of all code that compiles. If a running program violates any one of myriad rules, all bets are off. The program might behave as intended, or crash, or corrupt priceless data, or serve an Internet villain. The computer might even catch fire—rogue software could literally fry the original IBM PC.¹³

By declaring that certain coding errors yield undefined behavior, language standards permit compilers to skip runtime checks and optimize aggressively. They also shift the burden of ensuring predictability onto the programmer. Unfortunately, undefined behavior arises in many ways; appendix J.2 of the C standard lists scores,² and C++ adds many more.³

This article surveys the most prominent pitfalls, presents examples from production software, and suggests practical ways to prevent and detect such bugs in serial code. An earlier *Queue* article by Hans-J. Boehm and Sarita V. Adve discusses undefined behavior in multithreaded software.¹

GUESSWORK

Physical intuition misleads some developers into believing they can predict the behavior of software that executes undefined operations: “If defective track derails a locomotive, the train will go *somewhere*,” they reason, concluding that we can know where. If pure reasoning can’t deduce the outcome, surely experiment must be definitive: “Like Schrödinger’s cat, undefined software exists in an indeterminate state only until we observe its behavior, whereupon *something* will happen.” Try it and see, says this mentality.

Predicting the behavior of undefined code, however, is a fool's errand. Experiments merely record what happened when undefined code ran in the past. They can't predict the future, when the code encounters smarter compilers, upgraded runtime support, new hardware, and seemingly irrelevant differences in the execution environment. Pure reasoning about source code is futile because it relies on language standards that explicitly deem undefined behavior unpredictable.

Moreover, astonishment may begin well *before* execution reaches an undefined operation. An ominous passage in the C++17 standard portends a looking-glass world where cause follows consequence: “[If an] execution contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (*not even with regard to operations preceding the first undefined operation*)” [emphasis added].³

UNDEFINED BEHAVIOR IN VITRO

The C/C++ functions in figure 1 contain four undefined operations that I've seen repeatedly in production code; comments indicate (unsound) behavioral predictions. Developers might naïvely expect suitable arguments to execute the `if` branches, with foreseeable consequences: `null_deref(1)` in figure 1a causes a segmentation fault; `div_by_zero(1)` in figure 1b causes a hardware trap; `uninit(1)` in figure 1c reads arbitrary “stack garbage”; and `overflow(INT_MAX)` in figure 1d causes signed integer addition to “wrap around” to a negative value.

FIGURE 1: **BOGUS PREDICTIONS, FILE Ex_I.c**

```
10 int null_deref(int x) {
11     int *p = NULL;
12     if (x) {
13         return *p + 13; // SIGSEGV
14     }
15     else {
16         return 47;
17     }
18 }
```

(a)

```
20 int div_by_zero(int x) {
21     if (x) {
22         return 13 / 0; // trap
23     }
24     else {
25         return 47;
26     }
27 }
```

(b)

```
29 int uninit(int x) {
30     int u;
31     if (x) {
32         return u + 13; // garbage
33     }
34     else {
35         return 47;
36     }
37 }
```

(c)

```
39 int overflow(int x) {
40     if (x + 1 < x) { // wrap
41         return 13;
42     }
43     else {
44         return 47;
45     }
46 }
```

(d)

Testing these predictions pulls you through the looking glass. Compiling with Clang 11.0.0 and calling each function with arguments meant to exercise both `if` and `else` branches shatters naïve expectations. Function `null_deref()` does not seg fault, `div_by_zero()` does not trap, and all four functions *always* return 47, regardless of the `x` argument. It's as though the `if` branches have gone missing—which indeed they have: The compiler emits assembly code utterly devoid of arithmetic operations, comparisons, branches, and unlucky numbers. All four functions compile to the same two-instruction sequence that returns the same constant. If this code be a cat, it is Cheshire, not Schrödinger, vanishing almost entirely and leaving only a lingering `return 47` to smirk at the bewildered programmer.

Mathematical cognoscenti might suspect the compiler of exploiting an obscure proof from the 1960s: A function that returns an `int` may always return 47, because all numbers are equal to 47.⁴⁷ Actually, the compiler reasons as follows: The C and C++ language standards decree that execution Shalt Not reach undefined code; the `if` branches in figure 1 obviously execute undefined operations; therefore, the `if` branches must be unreachable and may be omitted. Such elision is not unique to Clang—other compilers, including GCC, act similarly—and it’s perfectly legal according to the standards: As noted earlier, reachable undefined code voids the warranty, *“even with regard to operations preceding the first undefined operation.”*

To the standards, eliminating undefined code does no harm; in practice it may be positively beneficial. Just as circulation through a gangrenous limb can be fatal, executing a code path containing undefined operations can unleash mayhem. Fortunately, the compiler regards such code paths with the unsentimental eye of a Civil War field surgeon: Better for the patient to live with one good leg than die from a bad one.

Figure 2 shows that the extent of amputation isn’t limited to a single errant instruction; it is unbounded. Function `sumdiv()` is meant to compute the sum of an integer’s divisors, but variable `s` isn’t initialized. Incrementing `s` is therefore undefined, so the entire function may reduce to `return 47` (which is how GCC 10.2.1 compiles it). The logic eliminated from figure 2 could have been arbitrarily more complex. The compiler won’t eliminate opaque function calls, which may `exit()` and

FIGURE 2: SUM-OF-DIVISORS CODE, Ex_II.c

```

5 | int sumdiv(int a) {
6 |   int s; // uninitialized
7 |   if (a) {
8 |     for (int i = 1; i <= a; i++) {
9 |       if (0 == a % i) {
10 |        s += i; // undefined
11 |       }
12 |     }
13 |     return s;
14 |   }
15 |   else {
16 |     return 47;
17 |   }
18 | }

```

thereby render undefined operations unreachable. Any amount of ordinary logic, however, is fair game for the hacksaw. One little careless undefined operation invites the compiler to discard your magnum opus, your life's work, your most brilliant code, wrought at staggering cost in toil and tears—all gone in the blink of an eye, without

your knowledge or consent. As compilers get smarter, the amputations will grow larger.

Of course, the standards merely permit but do not require compilers to remove undefined code paths; compilers may do anything whatsoever with such paths. Again,

it is folly to try to predict or control the result when undefined code is compiled or executed.

TAKING ACTION

How should programmers respond to the threat of undefined behavior?

undefined behavior

haiku

*meaningless software
vanishes at compile time
programmer says “wuh?”*

Some fulminate against their compiler, insisting that undefined code should behave as bug authors intend.⁶ Such rants have not achieved the desired effect. On the contrary, compiler writers have given notice that they will continue exploiting the liberties granted by the language standards.⁹

Other developers disable compiler optimizations in the hope of obtaining predictable behavior from undefined operations, as some versions of the Linux kernel and some open-source programs have done.¹⁴ This approach relies on nonstandard, nonportable features of particular compilers, and it impairs performance. A similar approach is to specify outcomes for particular undefined operations. For example, GCC's `-fwrapv` flag ensures that signed integer addition wraps on overflow.¹² Such flags aren't portable, and they're blunt instruments affecting all arithmetic within reach. While it might be perfect for the handful of operations that motivated the programmer to use `-fwrapv`, wrap-on-overflow in other parts of the code may surprise the programmer as thoroughly as the outcome without `-fwrapv`.

The correct approach, of course, is to write software whose behavior is predictable according to the relevant language standards. Diligent programmers study standards carefully and leverage tools to keep undefined operations out of their code.

FLAGGING MEANINGLESS CODE

Modern compilers are among the easiest yet most powerful tools for finding undefined code, but getting the most out of a compiler requires skill. Here GCC is used

to illustrate key techniques; other good compilers offer similar capabilities.

Developers might expect GCC's `-Wall` flag to live up to its name, but it actually enables only a fraction of available warnings. `-Wextra` adds several more, but many must be requested individually. For example, neither `-Wall` nor `-Wextra` activate `-Wnull-dereference` checks. Furthermore, some warnings rely on optimization. For example, both `-Wall` and `-Wextra` warn about use of uninitialized variables, but GCC performs the required analyses only if optimization is enabled too.¹² Once upon a time one of the authors encountered a lucrative proprietary program that was compiled for testing with warnings but no optimization and was compiled for production with optimization but no warnings. Nearly 100 uninitialized-variable bugs had eluded detection for years in this code, which developers scurried to fix after learning to compile in a more informative way.

How do compiler warnings find the four bugs in figure 1? GCC 10.2.1 *always* warns about division by zero, regardless of optimization level or warning flags. It warns about uninitialized variable use when invoked with both optimization and `-Wall -Wextra`. Optimization and `-Wnull-dereference` together reveal the bug in figure 1a. A warning courtesy of `-Wsuggest-attribute=const` draws attention to the faulty overflow test in figure 1d.

Figure 3 illustrates an insidious bug that arises when responsibility falls through the cracks between modules. The function in figure 3a assumes that its caller has initialized data passed by reference. The caller in figure 3b, however, assumes that the *callee* will initialize it, so `main()`

FIGURE 3: BUG SPANNING TWO TRANSLATION UNITS

```
3 int xtu(int *p) {  
4     // assumes caller has  
5     // initialized *p  
6     return *p;  
7 }  
  
4 int main(void) {  
5     int a, b, c;  
6     // assumes callee will  
7     // initialize variables  
8     printf("%d %d %d\n", xtu(&a),  
9             xtu(&b), xtu(&c));  
10 }
```

(a) File Ex_IV.c

(b) File Ex_IV_main.c

uses variables `a`, `b`, and `c` uninitialized. Compilers normally consider translation units (`.c` files) separately, so they can't spot a bug like this, which doesn't fully reside in either translation unit. Fortunately, compiling *and* linking both files with GCC's `-fto` flag yields a warning as a side effect of link-time optimization.

The best way to prevent defects is to catch bugs early, automatically, and inexpensively via “compile-time hygiene.”¹⁰ In addition to building software in a debugging/testing mode and a production mode, add a third “maximum warnings” mode: Invoke `gcc --help=warnings` to get a complete list of available flags, each described fully in the manual.¹² Strive for a clean build with as many warnings enabled as possible; compile once with optimization disabled (`-O0`) and again with `-O2` or `-O3`. Try to catch bugs spanning translation units using features such as link-time optimization.

For bonus points, additionally employ a good lint-like tool or a static analyzer such as Coverity,⁵ which inspects code more deeply and looks for a wider range of bugs than most compilers. GCC and Clang also have static analyzers that go beyond ordinary compiler checks. For example,

GCC's `-fanalyzer` feature seeks use-after-free bugs.

Like cats contemplating baptism, some developers view compile-time hygiene as a way to purchase dubious hypothetical piety via clear and present ordeal. True, the first time legacy code builds with maximum warnings enabled, it usually spews a daunting torrent of diagnostics; some are spurious, and all require time to fix or squelch. Investing effort to clean up the build, however, typically fixes several real bugs. It's also a good way to learn more about the programming language, which the compiler understands better than most programmers. In the long run, maintaining a clean max-warnings build pays handsome dividends, and it's particularly easy if this invariant is enforced from the day a program is born.

CASE STUDY: REDIS

Compile-time hygiene proves its practical value when applied to thoroughly tested industrial-strength software. Redis is an open-source NoSQL database that is widely used in commercially important applications.¹¹ By default Redis builds with GCC's `-Wall`, `-Wextra`, and only a few other warning flags, but its build system makes it easy to add more.

Building Redis version 6.2-rc2 with nearly all GCC warning flags yields a baker's dozen warnings about `NULL` pointer dereferences; figure 4 shows one bug. The `for` loop on lines 333–354 terminates when pointer variable `ln` is `NULL`; it is dereferenced on line 370. This bug debuted in release candidate 5.0-rc4 (August 2018) and regular (noncandidate) release 5.0.0 (October 2018). It remains in release 6.0.11 (February 2021). It's easy for a human auditor

FIGURE 4: NULL POINTER DEREFERENCE IN REDIS, FILE defrag.c

```

325  /* Defrag a list of sds values and a dict with the same sds keys */
326  long activeDefragSdsListAndDict(list *l, dict *d, int dict_val_type) {
327      long defragged = 0;
328      sds newswds, sdsele;
329      listNode *ln, *newln;
330      dictIterator *di;
331      dictEntry *de;
332      /* Defrag the list and it's sds values */
333      for (ln = l->head; ln; ln = ln->next) {
334          ...
335      }
336      /* Defrag the dict values (keys were already handled) */
337      di = dictGetIterator(d);
338      while((de = dictNext(di)) != NULL) {
339          if (dict_val_type == DEFRAG_SDS_DICT_VAL_IS_SDS) {
340              ...
341          } else if (dict_val_type == DEFRAG_SDS_DICT_VAL_VOID_PTR) {
342              void *newptr, *ptr = dictGetVal(de);
343              if ((newptr = activeDefragAlloc(ptr)))
344                  ln->value = newptr, defragged++;
345          }
346          defragged += dictIterDefragEntry(di);
347      }
348      dictReleaseIterator(di);
349      return defragged;
350  }

```

to overlook a bug like this. Testing may fail to catch it as a result of poor test coverage—or because the compiler removes the undefined code path. The easy and foolproof way to catch such bugs is the `-Wnull-dereference` flag, which GCC has supported since April 2016.

Inspecting the 50-odd `-Wfloat-equal` warnings leads to an apparently deliberate divide-by-zero, which may be undefined depending on the language dialect and implementation. The code in figure 5 divides by zero on line 526 to accomplish what the standard `signbit()` macro could do correctly, portably, and succinctly.

Other warnings spotlight the sort of confusion that often accompanies bugs. For example, `-Wduplicated-branches` yields three warnings, two of which refer to the code in figure 6. The long and complex `else if` condition spanning

FIGURE 5: DIVISION BY ZERO IN REDIS, FILE `util.c`

```

524     } else if (value == 0) {
525         /* See: http://en.wikipedia.org/wiki/Signed\_zero, "Comparisons". */
526         if (1.0/value < 0)
527             len = snprintf(buf, len, "-0");
528         else
529             len = snprintf(buf, len, "0");
530     } else {

```

FIGURE 6: DUPLICATED BRANCHES IN REDIS

```

1734     } else if (extent_state_get(extent) == extent_state_muzzy ||
1735                ((*r_extent_hooks)->purge_lazy != NULL &&
1736                !(*r_extent_hooks)->purge_lazy(*r_extent_hooks,
1737                extent_base_get(extent), extent_size_get(extent), 0,
1738                extent_size_get(extent), arena_ind_get(arena))) {
1739         zeroed = false;
1740     } else {
1741         zeroed = false;
1742     }

```

(a) File `extent.c`

```

3175     handler->emitResultFromCBuffer(handler, ln->ele, sdslen(ln->ele),
3176     ((withscores) ? ln->score : ln->score));

```

(b) File `t_zset.c`

lines 1734–1738 of figure 6a serves no purpose because it triggers the same action as the final `else` (lines 1739 and 1741 are identical). This anomaly originated in a commit of over 1,500 changes to 23 files; bugs that infiltrate via large updates tend to accumulate if not automatically detected. A similar undersimplification, the ternary operator on line 3176 of figure 6b, probably reflects the programmer’s fatigue rather than intentions. GCC has supported `-Wduplicated-branches` since May 2017.

Activating and heeding compiler warnings beyond `-Wall` and `-Wextra` would have squashed all of these Redis

bugs—and several others not reported here—minutes after they were first written.

DRILLING DEEPER

In addition to warning about bugs at build time, compilers can also instrument programs to detect undefined behavior at runtime. Check out GCC's `-fsanitize` and similar features in Clang.⁴ Stand-alone tools such as Valgrind can catch memory errors and other kinds of bugs during testing. The main limitation of testing is that it depends on inputs to exercise code—if you think fixing compiler warnings is a chore, try writing test inputs that cover every execution path in a large program. And testing can't trigger failures on code paths that the compiler eliminated because they contain undefined operations.

Bits

The code examples in this column are available at https://queue.acm.org/downloads/2021/Drill_Bits_O4_sample_code.tar.gz. The source files are valid as either C or C++. Shell scripts compile and run each example. Another script compiles a simple program in “max warnings” mode. Precompiled assembly code is provided for the examples of figures 1 and 2.

Drills

Try a few exercises involving this column's example code, open-source projects, and language standards. You'll gain different insights from studying highly simplified code, real production code, and documents that only a language lawyer could love.

1. Rewrite the code in figure 1d to test for overflow without causing it.
2. Can you get informative warnings about the bugs in figure 1 from Clang? From the Intel or Microsoft compilers?
3. Write small examples of several undefined operations. Determine the simplest set of compiler flags that detects each.
4. Intel assembly language includes special “undefined” instructions called `UD`, `UD1`, and `UD2`.⁸ Does your compiler emit them? When?
5. Consider the undefined operations enumerated in the C17² or C++17³ standard. How many are statically detectable in principle? How many are detected by your compiler? How many are impossible to detect statically [e.g., because of undecidability]?
6. Go hunting for bugs in an open-source project. How many bugs can you find in one day?
7. Go fishing: Grep source code for words likely to occur in comments near bugs [e.g., *overflow*, *careful*, *check*, etc.]. Look for visual red flags [e.g., bizarre indentation]. Do these heuristics find any bugs?

<https://xkcd.com/45/>

Catch-22: Implementing memmove()

C language standards from 1989 onwards have mandated the `memmove()` library function, which copies `n` bytes from one memory region to another (possibly overlapping) region:

```
void *memmove(void *dest, const void *src, size_t n);
```

The effect is as though the source array were first copied to a temporary array and then copied from there to the destination. Smart implementations avoid using a temp array by copying directly, “forwards” or “backwards” depending on whether/how the given arrays overlap; see the exemplary code in P.J. Plauger’s book, *The Standard C Library*.

Unfortunately, the C standards have decreed that pointer arithmetic and pointer comparisons *across different arrays* are undefined. This restriction, rooted in the Intel [x86 segmented memory](#) model of the 1980s, precludes writing `memmove()` the sensible way in portable well-defined C code. Plauger’s code performs an undefined comparison when the given arrays are distinct, and hence is not technically portable.

It’s easy for experts to run afoul of the restrictions on pointer operations. Steve Maguire’s book, *Writing Solid Code*, provides a memorably ironic example. Unlike `memmove()`, standard `memcpy()` yields undefined behavior when given overlapping arrays. Maguire shows how to implement a cautious `memcpy()` that tries to avoid undefined behavior by using an assertion to check for overlap. Unfortunately, the pointer arithmetic and pointer comparisons in the assertion are themselves undefined.

The books by Plauger and Maguire are excellent, and their aforementioned code would be, in a sane world, beyond reproach. The C language standards committee is aware of this situation and has [considered changes](#), e.g., declaring the outcome of far-flung pointer operations *implementation-defined* rather than *undefined*.

Acknowledgements

C++ Standards Committee member Hans Boehm of Google provided valuable feedback on an early draft.

References

1. Boehm, H.-J., Adve, S. V. 2011. You Don't Know Jack about Shared Variables or Memory Models. *acmqueue* 9(12); <https://queue.acm.org/detail.cfm?id=2088916>.
2. C17 language standard [N2176]. November 2017; https://web.archive.org/web/20181230041359/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf. The most official version is behind a hefty paywall: <https://www.iso.org/standard/74528.html>.
3. C++17 language standard [N4660]. 2017; <https://web.archive.org/web/20170325025026/http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4660.pdf>. Though they use different words, the C and C++ standards take nearly identical positions on undefined behavior.
4. Clang Compiler User's Manual. 2021; <https://clang.llvm.org/docs/UsersManual.html>.
5. Coverity Scan. 2021. Synopsys; <https://scan.coverity.com/>.
6. GCC Bugzilla. 2007. Debate over GCC optimization, [alleged] bug 30475; https://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475.
7. Hoare, C. A. R. 1981. The emperor's old clothes. 1980 ACM Turing Award Lecture. *Communications of the ACM* 24(2), 75–83; <https://doi.org/10.1145/358549.358561>.
8. Intel Corporation. 2020. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. Order Number 325462-073US, 2640, Table A-1; <https://software.intel>.

com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf.

9. Lattner, C. 2011. What every C programmer should know about undefined behavior. The LLVM Project Blog; <https://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html>.
10. Maguire, S. 1993. *Writing Solid Code: Microsoft Techniques for Developing Bug-Free C Programs*, 202. Microsoft Press.
11. Redis. 2021; <https://redis.io>; <https://github.com/redis/redis>.
12. Stallman, R. M., et al. 2020. *Using the GNU Compiler Collection*. GNU Press; <https://gcc.gnu.org/onlinedocs/gcc-10.2.0/gcc.pdf>.
13. van der Linden, P. 1994. *Expert C Programming: Deep C Secrets*, page 15. Prentice Hall .
- 14 Wang, X., Chen, H., Cheung, A., Jia, Z., Zeldovich, N., Kaashoek, M. F. 2012. Undefined behavior: What happened to my code? In *Proceedings of the Asian-Pacific Conference on Systems*, 9; <https://dl.acm.org/doi/10.5555/2387841.2387850>.
47. Wikipedia. 47; [https://en.wikipedia.org/wiki/47_\(number\)](https://en.wikipedia.org/wiki/47_(number)), <https://web.archive.org/web/20060901185414/http://www.pomona.edu/welcome/trek/47trek.shtml>.

TERENCE KELLY (tpkelly@acm.org) has battled undefined behavior both in theory and in practice, lately with Weiwei Gu and Vladimir Maksimovski of the University of Rochester. Like cats contemplating baptism, the authors look forward to submitting fixes for several of the diagnostics issued by their max-warnings build of Redis.

Copyright © 2021 held by owner/author. Publication rights licensed to ACM.