N3128
Date: 2023-05-05
Title: Taming the Demons -- Undefined Behavior and Partial Program Correctness[1]
Author: Martin Uecker

1. Introduction

There is no complete and universally-accepted consensus about the interpretation of undefined behavior (UB) in the standard. A popular interpretation states that a program becomes completely invalid as soon as there is potentially an operation with undefined behavior at run-time. We refer to this as the **abstract interpretation** of UB, because UB is considered as an abstract condition that directly invalidates the whole program. There is also a **concrete interpretation** that undefined behavior at run time simply allows any arbitrary concrete behavior that is technically possible for the specific operation that has the undefined behavior.

2. What the C Standard Says

UB is defined in s3.4.3p4 as:

> **behavior**, **upon use of** a nonportable or erroneous program construct or of erroneous data, **for which** this document imposes **no requirements**

Notably, it explicitly mentions both nonportable and erroneous program constructs, and not just the latter, and refers to specific behavior "upon use of…" and not to the whole program. The C99 Rationale further explains the intention:

> The terms unspecified behavior, undefined behavior, and implementation-defined behavior are used to categorize the result of writing programs whose properties the Standard does not, or cannot, completely describe. The goal of adopting this categorization is to allow a certain variety among implementations which permits quality of implementation to be an active force in the marketplace as well as to allow certain popular extensions, without removing the cachet of conformance to the Standard.

And then specific to UB it continues with:

> 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.

From this, it seems that the use of program constructs with undefined behavior was considered normal for C programs ("C code can be non-portable"). Therefore, such programs are not *strictly conforming*, but could still be *conforming,* i.e. acceptable, to a conforming implementation (cf. 4 Conformance). The C++ standard – but not the C standard - makes it clear that operations with UB make the whole program undefined (and can affect previous observable behavior):[2]

> However, **if any such execution contains an undefined operation**, this document places **no requirement on the implementation** executing that program with that input (not even with regard to operations preceding the first undefined operation).

1 This paper is a first result of the work of the UB study group and the content was unanimously endorsed by it.
2 https://eel.is/c++draft/intro#abstract-5

3. Impact on Optimization and Existing Practice

Some cases of UB are exploited by compilers for optimization. In this context, it is often stated that UB allows the optimizer to assume that a certain condition can not occur. This can be derived indirectly from the standard in the following way: UB allows arbitrary behavior. In consequence, any behavior is also allowed that can be derived by an optimizer by using the assumption that the condition for UB can not happen.

When deriving some information from such an assumption, *e.g.* that a variable has a certain value, then this can impact code generation even before the operation that would have UB if the assumption is violated. This is sometimes referred to as *backwards time travel*.[3] In the following example, a compiler can assume that 'b' is non-zero and set 'x' unconditionally to 42.

Example (backwards time travel only affecting non-observable behavior):
https://godbolt.org/z/ccjaTvx6T

```
extern int x;

int f(int a, int b)
{
  x = b ? 42 : 43;
  return a / b;
}
```

Such optimizations are generally valid under both interpretations, but there is a difference once observable behavior is affected. In the concrete interpretation, the behavior in questions refers to the concrete "behavior, upon use of the program construct" and not to the whole program. This has an important consequence:

With the concrete interpretation of UB, optimizations can not affect observable behavior that occurs before an operation that has UB.

While the concrete interpretations seems to substantially limit the optimizations that can be performed relative to the abstract interpretation of UB, the difference is actually very small and does not seem to affect common optimizations (such as the one in the example above). First, we note that behavior that is not observable can be changed according to the "as if" rule even in strictly conforming programs. Second, we will give a generic argument why – in practice - the abstract interpretation does not permit additional important optimizations: In portable C code I/O is performed using function calls of the standard library. A compiler can generally not assume that a function returns to the caller, because the function could call 'exit', 'abort', 'longjmp', or enter an infinite loop. For this reason, it is never allowed for a compiler to use any information derived from an operation with potential UB that comes after a function call in a way that could affect observable behavior before the function call.[4] Consequently, observable behavior that is accessed via function calls (*i.e.* all except volatile accesses) can not be affected by time-traveling optimizations even when using the abstract interpretation. In principle, a compiler could use special knowledge about C library functions and use the fact those functions always return to the caller, but we are not aware of any compiler which does this (and it hardly seems worthwhile).

---

3 https://devblogs.microsoft.com/oldnewthing/20140627-00/?p=633

4 For the same reason do compilers have a architecture-specific notion of "potentially trapping operation" that can not be hoisted before function calls, e.g. division by zero or memory accesses not known to be safe.

Example (questionable backwards time-travel):
https://godbolt.org/z/rYh4xM4sW

```
int f(int a, int b)
{
  int x = b ? 42 : 43;
  fprintf(stdout, "%d", x);
  fflush(stdout);
  return a / b;
}
```

In rare cases compilers were observed to violate these rules, *i.e.* a compiler uses assumptions derived from UB of operations that occur after a function call to optimize code before the call. In the few cases we could identify, this optimization is performed also for functions that terminate the program and then leads to incorrect behavior even for strictly conforming programs.[56]

Example (invalid backwards time-travel):
https://godbolt.org/z/z57a153os

```
extern void g(int x);

int f(int a, int b)
{
  g(b ? 42 : 43);
  return a / b;
}
```

This leaves us with the only observable behavior that is not hidden behind functions calls, *i.e.* volatile accesses. Here, some compilers are known to reorder volatile operations with potential UB. For example, this can have the effect that volatile accesses that should happen before a division do not occur when dividing by zero.

Example (reordering of a trapping operation and a volatile access):
https://godbolt.org/z/sdhfjxcsx

```
volatile int x;

int foo(int a, int b, _Bool store_to_x)
{
  if (!store_to_x) {
    return a / b;
  }
  x = b;
  return a / b;
}
```

---

5 https://developercommunity.visualstudio.com/t/Invalid-optimization-in-CC/10337428?q=muecker
6 https://bugzilla.redhat.com/show_bug.cgi?id=520916

In general, it is not clear to what extent these optimizations are intentional or compiler bugs (as the example above) and we got contradictory information from different compiler developers. The *volatile* keyword is often considered to have implementation-defined behavior that can not be relied on in portable programs and not all compilers consistently treat volatile accesses as observable behavior as mandated by the ISO C Standard. This issue has been reported as a bug report for an affected compiler, but while fixes have been suggested for the compiler the resolution will probably wait for a clarification from WG14.

In summary, with exception of volatile accesses compilers are generally consistent with the stricter concrete interpretation.

4. The Case for the Concrete Interpretation

The C standard could be made consistent with both interpretations, but the concrete interpretation seems to be better aligned with the current wording: For example, note 1 to the definition of UB or various footnotes mention specific behaviors as possibilities for UB (termination, traps, run-time constraint handlers). Annex J has the title “Portability Issues”. Annex L defines the term *bounded undefined behavior* as “undefined behavior (3.4.3) that does not perform an out-of-bounds store.” which only makes sense if UB is understood as concrete behavior. In general, the concrete interpretation seems better aligned with the different uses of UB in the C standard. If instead the abstract interpretation is adopted, any such wording, examples, and usage of UB should be reconsidered and revised accordingly.

In some cases where UB is exploited for optimization, the concrete interpretation gives stronger guarantees: It requires that *observable behavior, i.e.* I/O (when not buffered) and volatile accesses that appears before an operation that has UB is required to occur as specified by the ISO C standard. As a consequence, one can still show **partial correctness** of programs that have UB at some point during the execution. For example, if one can show that a program that controls a nuclear power plant is correct until after the I/O that initiates the shutdown of the nuclear reactor in an emergency, then one can be sure that the shutdown happens even when a later routine that outputs a status message has a bug that causes UB. In general, any I/O transaction which is shown to be completed (observable) can not be affected by errors later in the program. This seems to be a desirable property! An example closer to the real world experience of most C programmers is that a debug message (or I/O that causes an LED to light up) that occurs before an erroneous operation in the source code can not be reordered or optimized away by a compiler. Similar considerations apply to showing correctness when UB is defined by some implementation to invoke fault handlers (traps), which should then not be reordered relative to observable behavior. Kernel developers expressed the opinion that such reordering across volatile accesses is problematic when writing device drivers. To address such issues in C++, there is a proposal that would introduce a barrier ‘std::observable’ that ensures that observable behavior before this barrier is not affected by subsequent UB.[7] In our opinion, it seems much simpler and safer to ensure that all observable behavior automatically has this property instead of requiring that users add extra annotations. Instead, function annotations as introduced by C23 could be used to indicate where such optimizations are safe.

7 https://open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1494r2.html

5. Conclusion

We propose to clarify that UB is always concrete so that the programmer can be sure that observable behavior must occur with the correct output even when there is a condition that leads to UB afterwards. We believe that the abstract interpretation does not enable additional useful optimizations while it removes useful guarantees about partial program correctness and makes it more difficult for programmers to reason about programs.

Suggested Change: Since the wording is already consistent with the concrete interpretation, only the addition of a clarifying note is suggested. In 3.4.3 add after the definition of UB:

Note 3 to entry: Any other behavior during execution of a program is only affected as a direct consequence of the concrete behavior that occurs when encountering the erroneous or non-portable program construct or data. In particular, all observable behavior, i.e. I/O or volatile accesses, shall appear as specified in this document even when there is a subsequent operation with undefined behavior in the execution of the program.

Acknowledgements: The author thanks David Svoboda, Eskil Steenberg, Miguel Ojeda, Dave Banhan, Clive Pygott, Martin Sebor, Philipp Klaus Krause, Viktor Yodaiken, Hans Boehm, Paul McKenney, and Robert Seacord for insightful discussions about this topic.