[N2769] Redefining Undefined Behavior

Author: Eskil Steenberg
Date:6/21/2021

Introduction:

Undefined behavior, is a hotly debated issue in the C community. In this proposal I want to address the definition of Undefined behavior (UB) and its impact on real-world implementations, and propose a change to this section of the C standard. As this is a complex and confusing subject, I will try to first explain the issues this proposal tries to address.

The problem:

Having the C standard not define everything, is useful for many reasons. It gives C implementations freedom, and limits the scope of the specification.

A C implementation can not be held responsible for all the things a user can do, Just like it makes sense for a car manufacturer not to be responsible, if a driver drives recklessly. However, if car manufacturers use this common sense rule, to deactivate airbags, power steering, anti-lock brakes, and other safety features whenever the driver passes the speed limit, in order to optimize fuel economy, then something needs to be done. C implementations have been given great power to handle UB, but have unfortunately not shown great responsibility, with this power.

Lets say the programmer writes the following:

```
if(a == b)
    printf("Hello");
```

The naive assumption of most programmers is that the printf will only execute if a is equal to b. Due to UB, this is not guaranteed. a or b may have wrapped, or may be pointers to different objects, or a number of other UB may have enabled the compiler to assume the state of a and/or b, in a way that may not reflect the state of the bits at execution.

In the eyes of many, the fact that they can write the above code, and not be sure that the printf is only and always executed when a and b are equal, makes C an untrustworthy language. This is a grave problem for C, its standing and many mission critical software written in C. The code transformations that cause this perception of an untrustworthy compiler/language may be (and in the vast amount of cases are) entirely conformant to the C standard. Yet, they continue to confuse users, because they conflict with the perception of the language, and what users empirically can deduct from conforming implementations.

Given the large number of possible implications and nuances of different instances

of UB, It is not the objective of this proposal to discuss the implications of specific UB, but rather discuss UB itself as a concept. (I leave it as an exercise for the reader to come up with as many reasons as possible why UB may cause the above if statement to do something unexpected)

The naive solution is to force any compiler to prevent these transformations and require the implementation to include the users instructions as-is. In short "The compiler should do what I tell it, and not mess around". This, I think most people with a deeper understanding of C and compiler optimizations will recognize is not a workable solution, so I won't discuss that as an option further in this proposal.

Background:

The definition of "Undefined behavior", in the C standard changed between C89 and C99.

The full text is as follows:

C89:

Undefined behavior – behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately-valued objects, for which the Standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

C99:

1   undefined behavior behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements2   NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

The key change is the word "Permissible" was changed to "possible". This means that while the C89 standard gives 3 permissible options for implementations to handle UB, C99 makes no such requirements, and therefore renders the 3 options merely as examples of possible behavior. While the permissible behaviors in C89 are broad, C99 has no requirements at all. It has been debated if the C89 text is in fact so broad, that it has the same meaning as the C99 definition, of not imposing any requirements at all. This would make the C99 change merely a clarification of the C89 definition. I do not agree with this interpretation, and I will now try to outline how I think the two differ. Even if one can disagree, and argue that they are the same, the spirit of C89s definition is different.

The C89 spec offers the opportunity for an implementation to end translation or execution with a diagnostic message, or to define and document an implementation/ platform specific behavior. These two options almost entirely solve the problem of UB. If the translation stops and an error message is given, then the programmer can address the problem and the issue of trust becomes moot. If the implementation defines a consistent and documented behavior, then the programmer can read the documentation and expect a consistent, empirically educable behavior. Again confidence in the language's behavior is restored. Unfortunately almost no implementations choose to implement these options afforded by the specification.

Example:

```
int *a = NULL;
*a = 0;
```

All major compilers, can at translation, detect the unavoidable UB in the above code, yet none of them choose the prudent course of action to stop translation and issue a diagnostic message, to inform the user of the detected issue. This reluctance to communicate what the compiler does and knows is at the core of the perceived untrustworthiness of Compilers and the C language itself. As compilers have found new ways to optimize, they are increasingly relying on UB, and therefore the perception is that C compilers have gotten less trustworthy as they no longer engage in naive implementations.

I will return to the option of issuing diagnostic messages, and platform defined behavior, but for most of the rest of this discussion, I will ignore these two options, and only talk about what happens if the implementation has chosen neither of them.

An interpretation of "Ignore the situation entirely"

If we compare the C89 and C99 definitions of UB, the key phrase to focus on becomes C89s "Ignore the situation entirely". Some people read this as equivalent to C99s "The standard imposes no requirements". I would argue that "Ignore the situation entirely", is more limited in scope. I believe that "ignore" can have 2 possible meanings, either we execute the instruction as-is and ignore if they make sense or have unintended consequences, or we simply do not execute the instruction. Lets look at an example that assumes an architecture with 32bit integers:

```
a = a << 33; /* UB, shifting too many bits. */
if(a == b)
        printf("Hello");
```

In this case "ignoring the situation" could constitute shifting 33 bits with unknown results, or removing the shift entirely (One could imagine a hardware architecture where there are no instructions to fulfill this task). What "ignoring the situation" does not allow, is for the implementations to make any assumptions about the value of a after the shift. The implementation would therefore not be able to use the UB shift, to optimize the comparison. A similar example would be reading an uninitialized variable:

```
int x;
a += x; /* UB, reading uninitialized value. */
if(a == b)
        printf("Hello");
```

Again "ignore the situation", could either mean read x, with indeterminate results, or do not read x (Again one could imagine an architecture where it is not possible to read x, because it does not exist until initialized). In either of these cases the implementation would not be permitted to reason about the possible values of a in order to optimize the compare and branch. The implementation has to "ignore" the implications of the UB, it can't actively make assumptions about it.
This reading of "ignore the situation", differs quite starkly, from the current situation, where a in both examples would be "tainted" with UB, and the implementation would be free to make assumptions about its state in later operations. This "viral" nature of UB is course for much of the concerns surrounding UB. As UB spreads, it can influence code far from the initial UB, making debugging very difficult for the user.

However, "Ignoring the situation" could in other cases still cause some unexpected behavior. Consider the following code:

```
void function(int a)
{
        if(a + 1 < a) /* signed integers have undefined overflow */
                printf("Overflow!");
}
```

In this case, the compiler can "ignore" the possibility of overflow, and deduce that the if statement can never be true. This would be entirely permissible with C89s definition of UB. To differentiate this from previous examples, I here forth refer to this as "assumed absence of UB". While issues arising from assumptions of UB are not uncommon, issues arising from "assumed absence of UB" are more common, and harder to grasp. A revert to C89 definition of UB, would change permissible behaviors for UB, but would not in any way change what an implementation can do regarding "assumed absence of UB". Assuming the absence of UB Is very much in line with "Ignore the situation entirely". An example of "assumed absence of UB" optimization would be:

```
if(a == NULL)
        post_error_and_exit();
*a = 0;
```

The implementation can deduce that all branches lead to accessing the object pointed to by a, therefore it can assume a is not NULL, since that would be UB. With that assumption it can at compile time assume the result of the compare and remove the if statement and call to post_error_and_exit entirely. The safety and security implications of this optimization behavior are obvious. While this optimization behavior may look scary to the average developer, it's important to know that the assumed absence of UB, is a very powerful concept for compiler optimizations, and therefore has great value. An example would be multiplying a variable with a power

of 2, can be optimized to a much faster, and equivalent bit shift, if overflow is undefined. Simply changing the specification in such a way that implementations can't assume the absence of UB, would not be a workable solution either.

Shrödingers undefined behavior.

Another issue contributing to the view that C is a untrustworthy language, is that while an implementation has the right to define a undefined behavior, many implementations choose to not "officially" define a behavior, even when an implementation conforms to other standards, such as ABIs that do define a behavior. This means that a behavior can be both defined and undefined at the same time. Code may depend on an ABI that defines a behavior, while at the same time not be able to rely on that behavior because it remains undefined from a language perspective. This obviously also contributes to the confusion users are experiencing.

UB is and must be implementation specific.

Whatever changes are made to the definition of UB, we need to be cognizant that even trivial UB may not be detected by an implementation. We do not want to impose heavy requirements of implementation to make deep analysis of possible UB, and in many cases UB cant be detected by an implementation. How an implementation deals with UB is inherently implementation specific. Several projects exist to build external software to detect possible UB issues in C code. The C standard and everyone in the C community should obviously support such efforts. While these projects can improve code quality, and point out potential issues, they can not tell the user what their particular implementation will do. Only the implementation itself can know that. Given that compiler design is complex, and that each platform and implementation can itself define behaviors that the user relies on, the solution has to come from requirements of the implementations. Most developers are more concerned with how their code is transformed by the implementation they use, rather than how their code could potentially behave running in an abstract machine, described in a specification.

Proposed changes:

Undefined behavior – behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately-valued objects, for which the Standard imposes no requirements.

Permissible undefined behavior are:

0 Ignoring the situation completely with unpredictable results. It is not permissible to make assumptions about the results.

1 Behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message).

2 Terminating a translation or execution (with the issuance of a diagnostic message).

If a program construct at translation is determined to always result in undefined behavior, a diagnostic message must be issued.

An implementation may assume that a program will never get into an execution state with undefined behavior (unless the behaviour is defined by the environment), but must issue a diagnostic message if flow-control changes due to this assumption.

Discussion about the meaning of the proposed changes:

This proposals text offers a more restrictive definition of UB, by again using C89s "Permissible" definition rather than C99s "Possible" definition, but also goes further by more clearly specifying what is Permissible.

It adds the provision that an unpredictable result really must be unpredictable and that an implementation can not use UB as a predictor, unless it defined this behavior in a documented manner. This should stop a significant portion of the "virality" of UB.

Sections 2, and 3 remains unchanged.

A new provision requires implementations that find unavoidable UB, to communicate this to the user. This only applies to implementations that do UB analysis and should therefore not impact naive implementations. The final provision maintains, and clearly spells out an implementation's right to assume that a program will not enter on to a UB state, but it also requires the implementation to notify the user when this results in a changed branch.

The goal, is to let the compiler aggressively optimize, while at the same time keeping the user informed of its actions, and in the cases where it can't optimized due to the more restrictive definition of UB, be able to communicate to the user why its not able to do so.

It should be noted, that any existing implementation could choose to conform to these changes, either by changing of behavior, or specifying and documenting existing behavior or a mix there of.

Open Questions:

The proposed specification mentioned that a diagnostic message needs to be issued, if changes are made to flow-control. This is a somewhat arbitrary limitation. One could argue for other definitions of what transformations mandate a diagnostic message.
In the current state "ignoring the situation" could still be interpreted as "do nothing at all". Ideally we would like to have "ignoring the situation" only mean execute the instructions even if they make no sense, and then mandate that if that is not possible the implementation has to choose one of the other two options.

Appendix, Examples:

In all the following examples, the implementation has the option to terminate the translation or execution, or behave in a documented platform defined manner. The following is a discussion of what is permissible, if implementations do NOT choose either of these options.

Example 1:

```
int a[1], b;
b = x;
a[1] = FALSE; /* out of bounds write */
if(b)
        printf("hello");
```

The implementation can not remove the if statement, because it can not make any assumptions about the results (overwriting b) of the out of bounds write.

Example 2:

```
int a;
if(a == b) /* UB, reading uninitialized value. */
        printf("Hello");
```

The implementation can not remove the if statement, because it can not make an assumption about the results of reading an uninitialized value. If the implementation detects that this code will result in the undefined behavior of reading from an uninitialized value, it has to issue a diagnostic message.

Example 3:

```
if(a == NULL)
        post_error_and_exit();
*a = 0; /* paths lead to writing to what a points to. post_error_and_exit does not have noreturn qualifier. */
```

An implementation can assume that code will not get into a UB state and can therefore assume that a cant be NULL, and can therefore remove the branch and call, but it can no longer do so without issuing a diagnostic message.

Example 4:

```
int a[10];
a[x] = 0;
if(x >= 10)
        printf("Hello");
```

Again a implementation can assume that code will not get into a UB state and can therefore assume that a can't be 10 or more, and can therefore remove the branch and call, but it can no longer do so without issuing a diagnostic message.

Example 5:

```
int *a = NULL;
*a = 0;
```

If the implementation detects that this code will result in the undefined behavior of writing to NULL, it has to issue a diagnostic message.

Example 6:

```
a = a << 33; /* UB, shifting too many bits. */
if(a == b)
        printf("Hello");
```

The implementation can not make any assumptions of the value a after the undefined operation and can therefore not optimize away any of the following code.