N3534 - Draft - Undefined Behavior in the C Programming Language

by David Svoboda svoboda@sei.cmu.edu

This is a paper that describes what is currently known about Undefined Behavior in C. The paper was created by the Undefined Behavior Study Group, in hopes of publication.

We would therefore request a vote to form an editorial group to adjust this paper for eventual publication as a white paper.

Educational Undefined Behavior White Paper

Undefined Behavior

This educational document explains the concept of *undefined behavior* as used in the C Standard. This document clarifies the meaning of the term and its implications to C language programmers.

The C Standard [ISO/IEC 9899:2024] defines undefined behavior in Section 3.5.3 as:

> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this document imposes no requirements
>
> Note 1 to entry: 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).
>
> Note 2 to entry: J.2 gives an overview over properties of C programs that lead to undefined behavior.
>
> 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 (5.1.2.4) appears as specified in this document when it happens before an operation with undefined behavior in the execution of the program.

The C Standard allows for certain behaviors to be undefined. Undefined behavior allows a platform to either define platform-specific behaviors or ignore the possibility of an erroneous state. The language does not require a platform to diagnose these errors.

Undefined behavior is used in many places in the C standard and for various reasons such as:

- **Simplicity:** C is a relatively small and widely implemented language partly because implementations are not required to provide specific guarantees for undefined behaviors.
- **Performance:** Declaring some behaviors to be undefined allows for better performance. Requiring one specific behavior might penalize a hardware architecture that behaves differently.
- **Detectability:** Many problematic states that a program can enter are difficult or expensive for the implementation to detect. Undefined behavior shifts the burden onto the programmer to guarantee their program will not enter such a state.
- **Extensibility:** Where a behavior is not defined, a platform can choose to implement a specific extension of the language that adds additional features. Many compilers add extensions in one form or another.
- **Historical reasons:** Some behaviors cannot be defined today because platforms have implemented divergent choices that existing software relies on. Therefore, defining these undefined behaviors would break some existing platforms and the programs that rely on them.

Undefined behavior can either be explicitly specified in the standard or remain implicit if the standard does not define a behavior. The ISO C working group's charter [n3280] has, as an explicit goal:

```
Undefined behaviors, unspecified behaviors, implementation-defined behaviors,
and other portability issues enumerated in Annex J of the Standard should be
eliminated or reduced.
```

An addition to reducing undefined behavior, the committee has the additional task of maintaining the list of explicitly-specified undefined behaviors, as shown by Annex J.2.

Additionally, there are paragraphs in the standard where it is unclear whether a behavior is defined or not. This can mean that some platforms treat a behavior as defined while others treat it as undefined. For example, consider paragraph 17 of section 6.7.3.2, which describes structure and union types:

```
Within a structure object, the non-bit-field members and the units in which
bit-fields reside have addresses that increase in the order in which they are
declared. A pointer to a structure object, suitably converted, points to its
initial member (or if that member is a bit-field, then to the unit in which
it resides), and vice versa. There may be unnamed padding within a structure
object, but not at its beginning.
```

It could be argued that this paragraph requires that the first member of a struct must have the same pointer address as the struct itself. This argument is not resolved, because the Standard provides no clearer text.

As another example, consider C99 [ISO/IEC 9899:1999], section 6.5.5 paragraph 6, which explains the behavior of the modulo operator `%`:

```
When integers are divided, the result of the / operator is the algebraic
quotient with any fractional part discarded.90) If the quotient a/b is
representable, the expression (a/b)*b + a%b shall equal a.
```

This paragraph neglects to address the corner case of `INT_MIN % -1`. In this case, the quotient `INT_MIN / -1` is not representable as a signed int, as it is mathematically `INT_MAX + 1`, which is outside the range of signed ints. Consequently, `INT_MIN % -1` was considered to implicitly be undefined behavior, even though the mathematical answer is 0. This was made explicit in C11 [ISO/IEC 9899:2011] :

```
6When integers are divided, the result of the / operator is the algebraic
quotient with any fractional part discarded.105) If the quotient a/b is
representable, the expression (a/b)*b + a%b shall equal a; otherwise, the
behavior of both a/b and a%b is undefined.
```

Other types of behavior in the C standard.

Beyond undefined behavior, the C standard defines a range of terms related to behavior. Unlike undefined behavior, each of these terms define a constrained behavior where the implementation has some form of responsibility to uphold, even if it may vary between implementations.

- **Implementation-defined behavior:** The behavior is chosen by the implementation. Unlike Undefined behavior, a conforming implementation must choose a dependable behavior and document this behavior, so that it can be relied on by programmers.
- **Unspecified behavior:** Behavior where the standard offers 2 or more options for implementations to choose from.
- **Locale-specific behavior:** A behavior that depend on local conventions of nationality, culture, and language that each implementation documents.
- **Constraint:** This is a restriction of the language, either syntactic or semantic. It usually results in a compilation error.
- **Runtime-constraints:** A constraint that defines limitations of the C standard library. These are limitations that can apply to how the program uses the standard library and are therefore usually encountered at run time.
- **Diagnostic message:** Is when the standard mandates that a conforming implementation issue a message to the programmer. Usually this happens in the form of a warning or error.

All of these are different from Undefined Behavior in that, while they may produce different behaviors on different implementations, they do represent behaviors that a programmer can depend on, in a ISO compliant C implementation.

Undefined Behavior that is Assigned a Implementation-Defined Behavior

The C standard states that any platform is free to detect undefined behavior and to provide platform-specific behavior and document this behavior if it wishes. In this sense, what is in strict ISO C terms "undefined behavior" may be well-defined behavior on a particular implementation.

This can be very useful, because it enables implementers to extend C's capabilities, and thereby grants programmers access to platform-specific features. While the C language is designed to enable cross-platform development, programmers are free to only support a limited set of platforms. For example, there are implementations of C that do define the behavior of out-of-bounds array writes, signed integer overflow, and dereferencing null pointers.

For brevity, unless otherwise noted, this document will consider undefined behavior only in cases where the implementation has not defined a platform-specific behavior or implementation-specific behavior.

Detecting undefined behavior

Consider the following code:

```
int a[5];
a[x] = 0;
```

What should happen if x is 42? A language design could issue an error, exit the program, or resize the array, among other choices. However, any of these choices would require the implementation of the language to perform a test to see if the value is within the valid range of the array.

```
int a[5];
if (x < 0 || x >= 5) {
  /* Handle out-of-bounds write */
} else {
  a[x] = 0;
}
```

This range check would add work for the compiler and execution environment. Adding any requirement to detect if the assignment is out of bounds would come at a cost in run time performance, and complexity. Not only would the implementation have to check each access to the array, but it would also have to keep track of valid array ranges.

C is designed to be fast, simple, and easily implementable; this is why C does not require any detection of out-of-bounds states. Consequently, C cannot define a behavior for a state that isn't detected. The behavior must be undefined.

It is a common misconception that all undefined behavior in the standard stems from oversights, or from the ISO C working group's failure to agree on an appropriate behavior.

The above example clearly shows that it is not practical to define any consistent behavior for out-of-bounds array access without imposing considerable burden on the implementation to detect the state. The cost of detecting an erroneous state prevents the language from defining any behavior should it occur.

Furthermore, if the standard were to require a program to exit on an out-of-bounds write, then the following piece of code would become a valid way to exit a program:

```
int a[5];
a[24] = 0;
```

This is not a good way to deliberately exit a program. It is preferred that a program exit in a manner that the standard explicitly documents as exiting, such as by calling a function named `exit`.

Reconsider this code:

```
int a[5];
a[x] = 0;
```

Another interpretation of the above code is that if there are no requirements for an implementation to handle an out-of-bounds access, then the code contains an implicit contract that x can only be between 0 and 4. The implementation can then assume that the programmer is aware of the contract and consents to it, even if the implementation cannot by itself determine that the contract is valid by analysis of the possible values x may hold. The implementation therefore need not check the value of x.

If the programmer cannot guarantee that x is within range, they can rewrite the code:

```
int a[5];
if (x >= 0 && x < 5)
  a[x] = 0;
```

One big reason that many behaviors are undefined is that detecting these undefined behaviors may be difficult to do at compile time, or it may impose too much of a performance penalty at run time.

The existence of undefined behavior implies conversely that when a program has no undefined behavior, its behavior is well-specified by the ISO C standard and the platform on which it runs. This is a promise or contract between the ISO C standard, the platform, and the programmer. If the program violates this promise, the result can be anything, and is likely to violate the programmer's intentions, and will not be portable. We will call this promise the "Promise of Absence of undefined behavior".

A C program that enters a state of undefined behavior can be considered to contain an error that the platform is under no obligation to catch or report, and the result could be anything.

Promise of Absence of Undefined Behavior: the Contract between the Standard, Programmer, and Implementation

Consider this code:

```
x = (x * 4) / 4;
```

From a mathematical perspective, this operation should not change the value of x. The multiplication and the division should cancel each other out. However, when calculated in a computer, `x * 4` may result in a value that may not be expressed using the type of x. If x is an unsigned 32-bit integer with the value 2,000,000,000 and it is multiplied by 4, the operation could wrap on a 32-bit platform and produce 3,705,032,704. The subsequent division by 4 will then produce 926,258,176. Since the standard declares that operations on unsigned integers have defined wrapping behavior, the two operations do not cancel each other out.

If we instead perform the same operation using signed integer types, things might change because signed integer overflow is undefined behavior. By using a signed integer, the programmer has agreed to the contract that no operations using the type will ever produce overflow. Therefore, the optimizer is free to ignore any potential overflow and can assume that the two operations cancel each other out. This mean that there is a significant optimization advantage in declaring that signed integer overflow is undefined behavior.

The promise that the program contains no undefined behavior is a powerful tool that compilers can employ to analyze code to find optimizations. Consider:

```
int a[5];
a[x] = 0;
```

If x is any value below 0 or above 4, the assignment has undefined behavior. On many platforms, `a[-1]` and `a[5]` would be assigned to addresses outside the bounds of a. Without requiring implementations to explicitly add bounds checks, it becomes impossible to predict the side effects of an out-of-bounds write. The implementation is therefore allowed to simply ignore the fact that undefined behavior can happen. By writing the above code, the programmer respects a contract with the compiler that x will never exceed the bounds of the array.

If we consider:

```
int a[5];
a[x] = 0;
if (x > 5) {
  // ...
}
```

In this case, because the assignment has undefined behavior if x is not between 0 and 4, the `if` statement can be translated as if the condition were always false. This allows the compiler to optimize away the `if` statement entirely. The reason is that any behavior which

is changed by this optimization occurs when x is not between 0 and 4; as any behavior is allowed in this case, the optimization is allowed. This completely conforms to the standard, but it removes some predictability of undefined behavior and can make programs with undefined behavior much harder to debug. The out-of-bounds write no longer causes a predictable wild write and it also causes an `if` statement to be removed. Note that this optimization can occur only if the then-clause has no observable behavior, such as printing the value of x.

A common bug is to try to detect and avoid signed integer overflow with code like this:

```
if (x + 1 > x) {
  x++;
}
```

Because the `if` condition can only be false if there is undefined behavior, it can be translated as if it were always true. Consequently, many compilers will optimize away the `if` statement entirely.

The confluence of undefined behavior and more aggressive but standards-compliant compiler optimizations exposes latent bugs that may otherwise behave according to programmer intentions. These bugs are characterized as hard to find and diagnose. These bugs often do not appear at lower optimization levels. This means that such bugs do not appear in executables that programmers produce during development. Consequently, these bugs can bypass many tests. Debuggers tend to operate on on executables compiled with lower optimization settings, where many of these issues do not show up. This makes it harder to find and fix these bugs.

An early example of a vulnerability arising from such aggressive optimization is CERT vulnerability 162289.

State of Undefined Behavior

A common consideration when discussing undefined behavior is the question of when undefined behavior is invoked. While some have argued that programs that are able to procure undefined behavior have no requirements whatsoever, it is the position of the WG14 Undefined Behavior Study Group that a program must first reach a state of undefined behavior before the requirements of the language standard are suspended. This view is shared by implementers, who have had a history of classifying instances where this isn't true as compiler bugs. [citation needed]

Consider the following:

```
int a[5], x;
scanf("%i", &x);
a[x] = 0;
```

In this example, a programmer-provided index is used to access an array of five elements. While this program may be bad form, it is well-defined until and unless `scanf` sets x to

outside the range of the array. The programmer has (implicitly) guaranteed that the index used to access the array will stay within the bounds of the array, but this guarantee is maintained outside of the program. Many programs depend on input strictly conforming to a set of requirements to operate correctly. While this may present safety and security issues, the programmers must weigh those considerations against other factors, such as performance. Even a strictly conforming program could enter a state of undefined behavior under some environmental circumstances. A program is only erroneous if it enters a state of undefined behavior. An implementation is not released from complying with the ISO C standard because undefined behavior is possible when executing that program; the implementation is released only once the program has entered a state of undefined behavior.

Observability

A core tenet of the C standard is the "as-if" rule. This rule states that an implementation is not required to operate in the way the program is strictly written, so long as the implementation's observable behavior (defined in C23, Sub-clause 5.1.2.3, paragraph 6) is identical to the program. The program must behave, but not operate, as if the written program were executed.

This means that the actual program behavior can vary radically depending on how an implementation is able to transform the program, as long as its observable behavior remains constant. For example, two non-observable operations can be reordered. Consider:

```
int a, b;
a = 0;
b = 1;
```

These are two non-observable assignments (because neither `a` nor `b` is `volatile`). As two independent operations they are not required to be executed in any particular order. They may in fact be executed concurrently. If we then consider:

```
*p = 0;
x = 42 / y;
```

These two operations are also non-observable operations but both operations can produce undefined behavior (either by `p` pointing to an invalid address, or `y` producing a divide by zero). Because the operations are non-observable, they may be reordered. The "as-if" rule permits a platform to execute the division before the assignment, as long as the re-ordering is not observable. If `y` is zero, there is no guarantee that `*p` is written before the program enters a state of undefined behavior.

Time Travel

Because any non-observable operation can be reordered and transformed, a program might reach a state of undefined behavior in an ordering not explicitly expressed in the

source code. Due to the *as-if* rule, statements can be changed by undefined behavior before any actual undefined behavior is encountered during program execution. Consider:

```
int a[5];
if (x < 0 || x >= 5)
   y = 0;
a[x] = 0;
```

The implementation can determine that there is undefined behavior if x is not between 0 and 4, and therefore the `if` statement can be removed as it only affects such cases. This cases an out-of-order behavior known as "time traveling undefined behavior", where a program bug causes unintended consequences before the undefined behavior is encountered during program execution. It is as if the undefined behavior traveled backwards in time from the array access to the `if` statement.

Time traveling undefined behavior is permitted if it does not interfere with observable behavior that occurs before entering a state of undefined behavior. Consider:

```
int a[5];
if (x < 0)
   y = 0;
if (x >= 5)
   printf("Error!\n");
a[x] = 0;
```

In this case, the call to `printf` is an observable event, and any re-ordering requires it to execute correctly unless it is preceded by a state of undefined behavior. The compiler is not permitted to optimize away the second `if` statement. The first `if` statement however has no impact on the observable behavior and can therefore be removed.

Note: Historically, there have been cases where time travel has impacted observable state (CVE-2009-2692, described by Goodin 2009). Implementers have generally considered these to be implementation bugs. To clarify that they indeed are bugs, the document [N3128 Uecker] was proposed and accepted for C23. It adds the non-normative 3rd Note that clarifies the issue in the standard.

Static Undefined Behavior

Consider this code:

```
int a[5];
a[42] = 0;
```

Every time this code runs, it will produce undefined behavior. The undefined behavior does not depend on any dynamic or external factors other than the code being executed. Let us call this type of undefined behavior as *static undefined behavior*, because it only depends on variables that are known at compile time. The set of static undefined behaviors is not well-defined because different implementations have differing abilities to detect undefined behavior at compile time. Consider:

```
int a[5];
if (x > 0) {
  y = 42;
} else {
  y = MAX_INT;
}
a[y] = 0;
```

This code also contains static undefined behavior but requires a more complex analysis to reach that conclusion. The term "static undefined behavior" denotes any undefined behavior that is not dependent on runtime state. An implementation is under no obligation to detect static undefined behavior, but if an implementation does detect static undefined behavior we have recommendations for how to proceed. Static undefined behavior denotes expressions that always produce undefined behavior even if it's not proven that the expression will ever be evaluated.

Undefined Behavior is Caused by Programmer Error

Note that this section applies only to undefined behavior that is not defined by a particular implementation.

Any statement that produces a state of undefined behavior (except for the `unreachable()` macro) is erroneous, unless an implementation has defined its own behavior for that statement. An implementation is under no obligation to detect any undefined behavior. If, however, the implementation doesn't detect static undefined behavior, it is free to assume the statement will not produce undefined behavior. Therefore, any static undefined behavior (again, excepting `unreachable()`) should be considered a programmer error and not an intended use of the language. In these cases, an implementation should produce a diagnostic message when it detects undefined behavior.

An implementation can assume that a program will not enter a state of undefined behavior, but no implementation should assume that a program that reaches a state of undefined behavior is intentional.

Consider again:

```
int a[5];
a[x] = 0;
```

The assignment may or may not produce undefined behavior. In this case if we follow the rule "assumed absence of undefined behavior", we can assume that x must be between 0 and 4. The assignment is an assignment, but it also provides a hint to the compiler as to what x may be. If we then add:

```
int a[5];
a[x] = 0;
if (x > 4)
  ...
```

If the `if` statement here produces no observable behavior, then it can be considered dead code and optimized away. The `if` statement doesn't produce undefined behavior, it just cannot happen without undefined behavior. Now, consider:

```
int a[5];
if (x > 4) {
  a[x] = 0;
}
```

Again, this code may or may not trigger undefined behavior, but if the assignment is ever executed it is guaranteed to trigger undefined behavior. (Note that an implementation is not required to detect the undefined behavior.) In other words, the undefined behavior is static, but only if the assignment is executed.

The correct interpretation of the detected static undefined behavior is that the code is erroneous. It is incorrect to interpret the above code as a valid way for the programmer to express that x is 4 or less. The "assumed absence of undefined behavior" rule only applies to the way a construct can be assumed to be executed; it should not be taken to mean that a construct that always produces undefined behavior will never be executed. One divided by X lets the compiler assume X is not zero, and X divided by zero should cause the compiler to assume unintended programmer error.

The one exception to this is the `unreachable()` macro. The `unreachable()` macro is the only way for a programmer to express that a statement can be assumed to never be executed. Incidentally, executing `unreachable()` is undefined behavior, but it should not be regarded as equivalent to other undefined behavior in this regard.

For example:

```
if (x > 4)
  unreachable();
```

This is a correct way to express that a compiler can assume that x is smaller or equal to 4. Despite `unreachable()` being undefined behavior, it is not equivalent to:

```
if (x > 4)
  x /= 0;
```

Division by zero is undefined behavior, but unlike `unreachable()`, it is assumed to be a programmer error. The `unreachable()` macro can therefore not be implemented by the programmer by producing undefined behavior in some way other than the `unreachable()` macro. undefined behavior is also erroneous even when it can be determined never to be executed. The following can be detected as erroneous:

```
if (0)
  x /= 0;
```

C is designed to make naive, as well as highly optimizing, implementations possible. The C standard therefore places no requirements or limits on the efforts an implementation

takes to analyze the code. Whichever erroneous undefined behavior may be detected will therefore vary between implementations.

Apparent Predictable Behavior of Undefined Behavior

Operating systems and even hardware have been designed to mitigate the side effects of unintentional undefined behavior, or deliberate sabotage using undefined behavior, with features such as protection of the memory containing the executable or execution stack. Due to some of these protections, some undefined behavior is predictably caught at run time. This mitigates the unpredictable nature of undefined behavior and improves the stability and security of the system. However, this can also give the false impression that some undefined behavior has predictable side effects. While dereferencing null pointers is technically undefined behavior, doing so is defined to trap on some platforms, including POSIX. Even if the behavior of dereferencing null is reliable on a platform, the compiler's assumption that the code will not dereference null will make it unreliable.

Some undefined behavior was initially included in the C standard because the standard wanted to allow for different platform designs. Over the years, some designs have grown so dominant that few programmers will ever encounter a platform that does not conform to these dominant designs. One example of this is two's-complement arithmetic, which causes signed integer overflow to wrap.

This means that many undefined behaviors have predictable behavior on most platforms:

Undefined Behavior	Convention
Dereferencing null pointer	Traps
Signed integer overflow	Wraps
Using the offset between 2 allocations	Treats pointers as integer addresses
Comparing the pointer to freed memory with a newly allocated pointer	Treats pointers as integer addresses
Reading uninitialized memory	You get whatever is there

Such behavior is not defined by the C standard but can seem to be predictable. Predictability is of great value to most programmers. The knowledge of how the underlying platform operates lets the programmer predict and diagnose bugs. A trapped null pointer dereference is easy to find in a debugger. In fact, a programmer may deliberately add a null pointer dereference to a program to invoke a core dump. In MSVC uninitialized memory is initialized to 0xCDCDCDCD, a pattern that is instantly recognizable for any experienced Windows programmer. If the sum of two large positive signed integers results in a negative value, a wise programmer will suspect signed integer overflow which happened to wrap.

This apparent predictability of many types of undefined behavior hides the fact that undefined behavior is not predictable. This causes many programmers to either not realize that some of these behaviors are undefined or confuse undefined behavior with

implementation-defined behavior. They may believe that undefined behavior is defined in the C standard and undefined behaviors may be non-portable, but they may assume that the behavior of their platform applies to all platforms, or other machines with same platform as their machine. This faulty assumption creates a variety of hard-to-diagnose issues that we will explore further.

An out-of-bounds write may have a wide range of consequences as it can disturb many kinds of state. However, most programmers would assume that an out-of-bounds write is executed as a write operation, which is not true in general. If we consider another undefined behavior such as signed integer overflow, it is even less predictable that a simple arithmetic operation can have a wide range of unpredictable outcomes.

Compiler restraint

Undefined behavior in C gives an implementation wide latitude to optimize the code. This freedom has enabled implementers to successively generate faster and faster machine code, which enables significant reduction in computing time and energy consumption for a wide range of workloads. C is the de facto benchmark for efficiency that other languages are compared against and strive to match.

Significant portions of undefined behavior, such as Aliasing, Provenance and Overflow are specifically designed to enable implementations to make optimizations. Violating these categories of undefined behavior is likely to cause unpredictable behavior only when an implementation engages with these opportunities to optimize code.

As many implementations support varying levels of optimizations, it is feared that compilers, at higher levels of optimizations, will ignore the C standard and break existing code. This is a misconception. Most C implementations are consistent with the C standard even at the highest levels of optimization settings. Optimizations often reveal existing bugs in the source code that are initially perceived as compiler bugs Regehr, *Is That a Compiler Bug?*. These bugs are usually in violation of the C standard even when the program operates consistently with the programmer's expectations.

The higher a level of optimization is employed, the more bugs are exposed, but as the code is further transformed, it also becomes harder to debug. Many tools like debuggers depend on low levels of optimizations to be able to correctly associate the binary's execution to the source code. This compounds the difficulty of diagnosing undefined behavior bugs.

Given the misconception that optimizations break code, rather than reveal latent bugs, implementers often unfairly get blamed for issues arising from undefined behavior. This has made many compilers avoid making certain optimizations, even when supported by the specification, if they anticipate a programmer backlash. For example, GCC, starting at version 4.2, introduced a `--fno-strict-overflow` flag which disabled optimizations that presumed that signed integer overflow is undefined behavior, based on the fact that allowing such optimizations broke many programs. This creates a gray area, where unsound code that contains undefined behavior may have an undocumented reliable or

semi-reliable behavior. This gray area comes at the cost of denying performance afforded by the standard to compliant code.

Safety and security

C is regarded by some as an *unsafe or insecure* language. However, a language specification can neither be safe nor unsafe in itself. While a language could be designed to satisfy some particular definition of safety or security in mind, it is only language implementations that can enforce safety. The C standard does not require an implementation to check for several types of errors, but it also does not prevent an implementation from doing so. Hence, each implementation may choose the level of safety guaranteed.

The most popular implementations of C choose not to make many guarantees such as memory safety but instead choose to prioritize performance and power efficiency. There are safer implementations, but these are predominantly used to detect issues during development rather than to add additional protections to deployment. One such implementation is Valgrind, whose default tool "memcheck" detects out-of-bounds reads and writes to memory on the heap, as well as uninitialized reads, use-after-free errors, and memory leaks. Valgrind achieves these safety constraints at a significant performance cost. Many different implementations such as GCC, LLVM and MSVS offer various tools for detecting and diagnosing undefined behavior. Several static analyzers also exist to alleviate this problem.

While not in any way mandated by the C specification, the prevailing modus operandi of C programmers consists of using safety-related tools to detect issues during development, rather than as backstops during deployment. A major drawback of this approach is that since undefined behavior is a state that often cannot be definitively detected until it occurs at run time, there is no easy way to definitively guarantee that a program will not enter a state of undefined behavior.

Despite this, it is worth noting that some of the most trusted software in the world, like the Linux kernel, Apache, MySQL, Curl, OpenSSL and Git are written in C. The simplicity of C makes it significantly easier to read and detect issues.

C does suffer when the standard is unclear, particularly in areas of the memory model and concurrent execution. Rules about aliasing, active type, thread safety, and volatile leaves a lot open to interpretation as to what is undefined behavior, and what is not. On many of these issues there is a lack of consensus within WG14. Most implementations do support behaviors that in the strictest reading of the standard would be considered undefined behavior simply because of programmer expectation, and to be able to compile important existing software. In this sense most implementation deviate from the standard, but how and how much they deviate varies. Some projects like the Linux Kernel, has explicitly opted out of these ambiguities and defined their own requirements.

Advice for Programmers

As this document has hopefully illustrated, undefined behavior in the context of C is complex. To simply say that certain guarantees have been omitted from the standard does not convey this complexity.

C is designed to be a language that trusts the programmer. In the case of undefined behavior, programmers should interpret this to mean "Trust the programmer not to initiate undefined behavior", rather than "The programmer can trust undefined behavior if they know the underlying implementation and platform". The Undefined Behavior Study Group therefore strongly advises programmers to avoid any undefined behavior, unless a platform has explicitly defined that behavior. Testing to determine what observable effect use of a non-portable or erroneous program construct has on your platform is insufficient cause for assuming the undefined behavior will consistently have the same behavior on all platforms, including the next one that your code will run on. Only trust an implementation's explicit documentation of a language extension that defines a behavior.

Advice for Implementers

We advise that implementations clearly document any language extensions that replace undefined behavior so that programmers can differentiate between such extensions and seemingly predictable but still unintended behavior. Finally, if an implementation can detect static undefined behavior, it should produce a diagnostic message (unless it provides a specific definition for that undefined behavior.)

A computer language is a tool for humans to communicate with computers, but it is also a tool for computers to communicate with humans. Humans spend more time reading the code they write and trying to figure out why its behavior does not match their expectations, than computers do.

Traditionally, implementations have been black boxes that programmers must rely on without understanding how they operate. This approach causes issues, because modern compilers do not operate like many programmers expect them to. We therefore recommend that implementations try to find ways to be more transparent with their transformations. The ability for programmers to inspect code that has been transformed could reveal out-of-order issues, code removal, load/store omissions and other non-obvious transformations. We recognize that this involves significant user interface and architectural challenges.

Acknowledgements

This white paper is the result of many invaluable discussions in the Undefined Behavior Study Group of ISO/IEC SC22 WG14. The following individuals, in particular, made significant contributions to this white paper: Eskil Steenberg Hald, David Svoboda, Christopher Bazley, Robert Seacord, and Martin Uecker.

References

[ISO/IEC 9899:1999] ISO/IEC. Programming Languages—C, 2nd ed (ISO/IEC 9899:1999). Geneva, Switzerland: ISO, 1999.

[ISO/IEC 9899:2011] ISO/IEC. Programming Languages—C, 3rd ed (ISO/IEC 9899:2011). Geneva, Switzerland: ISO, 2011.

[ISO/IEC 9899:2024] ISO/IEC. Programming Languages—C, 5th ed (ISO/IEC 9899:2024). Geneva, Switzerland: ISO, 2024.

[n3280] ISO/IEC. The C Standard charter. June 12, 2024