WG 14, N2365

# Defining Undefined Behavior

David Svoboda
svoboda@cert.org
Date: 2019-03-22

## The Problem

C17 defines many categories of undefined behavior (203 in this count). Many examples of undefined behavior (UB) are assumed to be well-defined by programmers, typically because their platform specifies the behavior. For example, signed integer overflow is undefined behavior. However, on x86 platforms, and, in fact, most platforms that use twos-complement arithmetic, signed integer overflow wraps; that is `INT_MIN == INT_MAX + 1 == -INT_MIN`. Consequently, many programs rely on undefined behavior, and their developers only learn that the behavior is undefined when porting their code to a new platform. This leaves developers in a nasty position: a large body of code that works properly only on platforms that make the same assumptions about certain undefined behaviors as the developers' platform.

This is not a new problem, it was popularized in the eighties as vaxocentrism. Today one can argue that Intel has replaced the VAX as a platform ubiquitous enough to fool many developers into incorrect assumptions of undefined behavior.

Furthermore, compilers and their optimizers can change how programs with UB actually behave. One of the first such instances, reported in 2008, was CERT Vul# 162289, where a compiler's optimizer silently removed bounds checks because they exhibited UB, even though without optimization the bounds checks performed as expected.

## Related Work

In document N2278, Yodaiken proposes that the definition of undefined behavior (in C17, Section 3.4.3) be constrained to prevent optimizers from changing working code. While this would plug many security holes, it would disable too many useful optimizations; a more fine-grained solution is required. Yodaiken makes a similar proposal focusing on aliasing in document N2279. This is a valid solution, but it addresses only one type of undefined behavior…many others remain un-addressed.

Several technologies exist to provide implementations for specific undefined behaviors. For example, GCC provides switches to guide the behavior of signed integer overflow. The `-ftrapv` switch causes any signed overflow in the translation unit to trap, while the `-fwrapv` switch causes signed overflow in the translation unit to wrap, assuming a twos-complement implementation of integers.

# Approach

We propose that ISO C should provide a standard framework for specifying common implementations of specific UBs. We will provide three examples of UB below and then suggest the general pattern and wording. Once this proposal is passed, we can add many more categories of undefined behavior.

The purpose of the framework is twofold. For each UB it addresses, it should allow the developer to explicitly state the expected behavior for one or more declarations. This informs subsequent developers of the assumptions, and it forces compilers to respect the expected behavior and not modify it during optimization. If the compiler cannot enforce the expected behavior, it can issue an error and exit.

Each UB implementation will have a mechanism to turn it on and off. While on, the compiler (assuming it supports the implementation) must not allow the optimizer to change the expected behavior, and so the compiler will produce safe, though inefficient, object code. The compiler is free to produce a fatal error diagnostic if it cannot support the expected behavior. When turned off, the compiler is free to modify undefined behavior as it pleases. Turning off a specific UB implementation might be done within a single translation unit to prevent optimization of one function while enabling it for others. Compilers that do not support per-function tweaking of such optimizations may issue a fatal diagnostic if the program tries to turn off a UB implementation in a translation unit.

We have debated whether to use macros, pragmas, or attributes to support our framework. Macros are easy to use, and developers can use the macros to ascertain the platform's capabilities. However, they are typically not visible to compilers. Pragmas are intended to be fed directly to compilers. There are a few standard pragmas, and we could define more. However, pragmas that are not recognized by a compiler are silently ignored (C17 s6.10.6p1). This means that a noncompliant compiler will fail to inform a developer if it does not support some particular implementation of a UB, and we would prefer such a platform to fail to compile code unless it is aware that it supports the required implementation. Attributes are not in C17, but are being proposed by N2269. They are used to apply to variables, functions, and declarations, whereas our framework is intended to apply to translation units, and, when supported, individual declarations.

We will therefore provide an API using macros, but the macros will use pragmas to communicate with the compiler.

For most undefined behaviors, we will provide one or more implementations that constrain the compiler to verify that that implementation is supported, and furthermore the compiler will respect that implementation. For each undefined behavior, we can also provide a default implementation that imposes no restrictions (e.g., the compiler need acknowledge no support and is free to optimize the UB as it pleases). For each UB, only one implementation (or the default) can be in effect at any time; setting one macro causes the others to be overridden. The 'default' behavior will be indicated by macros and pragmas that end with the word STRICT.

## *Example 1: Signed Integer Overflow*

The most common behaviors for handling signed integer overflow are wrapping and trapping, as supported by many modern platforms, including x86, and as enforced by GCC's `-fwrap` and `-ftrap` options. We will support both of these, first by the macros

```
#define __STDC_UB_SIGNED_OVERFLOW_WRAP  \
    #pragma STDC UB SIGNED_OVERFLOW WRAP
```

A platform that does not support signed integer wrapping can define it thusly:

```
#define __STDC_UB_SIGNED_OVERFLOW_WRAP  \
    #error "Signed Integer Wrapping not supported"
```

Likewise, platforms that do not support wrapping on individual functions (as opposed to the entire translation unit) can signal an error if the STDC UB SIGNED_OVERFLOW WRAP pragma appears after any function definitions.

Trapping can be supported by the following macro which supports a similar pragma:

```
#define __STDC_UB_SIGNED_OVERFLOW_TRAP  \
    #pragma STDC UB SIGNED_OVERFLOW TRAP
```

The optimizer could be constrained to not optimize code using the following macro:

```
#define __STDC_UB_SIGNED_OVERFLOW_PERMISSIVE  \
    #pragma STDC UB SIGNED_OVERFLOW PERMISSIVE
```

Finally, the optimizer may be granted full license to optimize signed integer overflow with the following macro and similar pragma:

```
#define __STDC_UB_SIGNED_OVERFLOW_STRICT  \
    #pragma STDC UB SIGNED_OVERFLOW STRICT
```

GCC currently supports wrapping and trapping for translation units only, so their pragmas would have to precede all function definitions, and the pragmas which implement the -ftrapv, -fwrapv, and -fstrict-overflow functionality. They can choose to implement per-declaration wrapping and trapping.

## *Example 2: Strict Aliasing*

Strict aliasing is mandated by ISO C, and can trip up many unwary programmers. Several compilers allow programmers to relax strict aliasing, permitting programs to violate it without

having their behavior changed by the optimizer. For example, GCC provides the -fno-strict-aliasing switch. The following macro and pragma can disable strict aliasing:

```
#define __STDC_UB_ALIASING_PERMISSIVE  \
    #pragma STDC UB ALIASING PERMISSIVE
```

To enable strict aliasing, and therefore have faster code, use:

```
#define __STDC_UB_ALIASING_STRICT  \
    #pragma STDC UB ALIASING STRICT
```

## *Example 3: Dereferencing Null*

In ISO C, dereferencing a null pointer is undefined behavior. A common convention among platforms is that memory location 0 is unreadable, and so dereferencing null will immediately trap. However this is not universal, and under some special circumstances a null dereference has been optimized away leading to a vulnerability documented by Dan Goodin in 2009.

A program that wishes to enforce that dereferencing null always traps can use the following macro:

```
#define __STDC_UB_NULL_DEREFERENCE_TRAP  \
    #pragma STDC UB NULL_DEREFERENCE TRAP
```

A program that wishes null dereferences to not be optimized away (regardless of what they actually do), can use the following macro. This is analogous to GCC's -fno-delete-null-pointer-checks switch:

```
#define __STDC_UB_NULL_DEREFERENCE_PERMISSIVE  \
    #pragma STDC UB NULL_DEREFERENCE PERMISSIVE
```

A program that wishes to allow the compiler to optimize any null dereferences can use the following macro. This is analogous to GCC's -fdelete-null-pointer-checks switch:

```
#define __STDC_UB_NULL_DEREFERENCE_STRICT  \
    #pragma STDC UB NULL_DEREFERENCE STRICT
```

## *Some Extra Features*

There are some useful extra features to add to our set. In particular, we wish the compiler to support a stack of changes that we make to undefined behavior implementations. Both MS Visual C++ and GCC support stacks in some of their platform-specific pragmas. This allows us to tweak the requirements, but reset them back to known values.

```
#pragma STDC UB PUSH
```

The above pragma pushes any specific UB settings onto the UB stack.

```
#pragma STDC UB POP
```

This pops any specific UB settings from the UB stack. If any UB settings were changed since the last PUSH, they are reset to previous values. If the stack is empty, then the UB settings are reset to whatever values were provided by default from the compiler. (The compiler may provide command-line options to tweak the settings.)

# Proposed Wording Changes

*Add behavior.h to the list of standard headers in p7.1.2p2.*

*Create a new section 7.31, to precede the current section 7.31 "Future library Directions". The text of the new section would be as follows:*

7.31 Implementations of Undefined Behavior

1. The header `<behavior.h>` defines several macros that suggest platform-specific definitions of behavior that are explicitly undefined in this standard. When supported, each macro expands to a pragma, which indicates a precise behavior that the program assumes is supported by the platform and translator. If a macro's corresponding pragma is not supported on a platform, the macro instead expands to an #error directive.
2. The translator also supports the following two pragmas.

```
#pragma STDC UB PUSH
```

This causes any current UB pragmas to be saved, so that they may be later restored.

```
#pragma STDC UB POP
```

This restores any specific UB settings that were previously saved by a UB PUSH pragma. If any UB settings were changed since the last PUSH, they are reset to previous values. If the stack is empty, then the UB settings are reset to whatever values were provided by default from the translator.

7.31.1 Signed Integer Overflow

1. These pragmas can be used to enforce specific behaviors when a computation involving signed integers results in overflow (that is, the mathematical result could not be represented in the appropriate signed integer type).
2. The pragma:

```
#pragma STDC UB SIGNED_OVERFLOW WRAP
```

constrains the translator to assume that signed integer overflow always produces wrapping. That is, if a mathematical result is not representable in the appropriate signed integer type, then it can be converted by repeatedly adding or subtracting 2 raised to one plus the number of precision bits until the result is representable in the given type.

3. The macro

```
__STDC_UB_SIGNED_OVERFLOW_WRAP
```

either expands to the corresponding pragma or expands to an `#error` directive.

4. The pragma:

```
#pragma STDC UB SIGNED_OVERFLOW TRAP
```

constrains the translator to assume that signed integer overflow always traps.

5. The macro

```
__STDC_UB_SIGNED_OVERFLOW_TRAP
```

either expands to the corresponding pragma or expands to an `#error` directive.

6. The pragma:

```
#pragma STDC UB SIGNED_OVERFLOW PERMISSIVE
```

constrains the translator to assume the platform has defined the behavior regarding integer overflow, and, while unspecified, the program depends on this behavior.

7. The macro

```
__STDC_UB_SIGNED_OVERFLOW_PERMISSIVE
```

expands to the corresponding pragma.

8. The pragma:

```
#pragma STDC UB SIGNED_OVERFLOW STRICT
```

relaxes any constraints the translator currently has regarding signed integer overflow. That is, the program makes no assumptions about what happens if signed integer overflow occurs.

9. The macro

```
__STDC_UB_SIGNED_OVERFLOW_STRICT
```

expands to the corresponding pragma.

7.31.2 Strict Aliasing

1. These pragmas can be used to enforce specific behaviors when data is accessed via a pointer of an incompatible type, as is explicitly forbidden by Section 6.5, paragraph 7.
2. The pragma:

```
#pragma STDC UB ALIASING PERMISSIVE
```

constrains the translator to assume the platform has defined the behavior regarding accessing data through incompatible types, and, while unspecified, the program depends on this behavior.

3. The macro

```
__STDC_UB_ALIASING_PERMISSIVE
```

either expands to the corresponding pragma or expands to an `#error` directive.

4. The pragma:

```
#pragma STDC UB ALIASING STRICT
```

relaxes any constraints the translator currently has regarding access to data from incompatible pointer types. That is, the program makes no assumptions about what happens if data is read or written via an incompatible pointer type.

5. The macro

```
__STDC_UB_ALIASING_STRICT
```

expands to the corresponding pragma.

7.31.3 Dereferencing Null

1. These pragmas can be used to enforce specific behaviors when a null pointer is dereferenced* This is undefined behavior, specified by footnote 106, in Section 6.5.3.2, paragraph 4.

*Footnote: This can is done by applying the *, ->, or array subscript operator ([]) to a null pointer.

2. The pragma:

```
#pragma STDC UB NULL_DEREFERENCE TRAP
```

constrains the translator to assume that the program should trap if a null pointer is dereferenced.

3. The macro

```
__STDC_UB_NULL_DEREFERENCE_TRAP
```

either expands to the corresponding pragma or expands to an `#error` directive..

4. The pragma:

```
#pragma STDC UB NULL_DEREFERENCE PERMISSIVE
```

constrains the translator to assume the platform has defined the behavior regarding null pointer dereferencing, and, while unspecified, the program depends on this behavior.

5. The macro

```
__STDC_UB_NULL_DEREFERENCE_PERMISSIVE
```

either expands to the corresponding pragma or expands to an `#error` directive.

6. The pragma:

```
#pragma STDC UB NULL_DEREFERENCE STRICT
```

relaxes any constraints the translator currently has regarding null pointer dereferences. That is, the program makes no assumptions about what happens if a null pointer is dereferenced.

7. The macro

```
__STDC_UB_ NULL_DEREFERENCE _STRICT
```

expands to the corresponding pragma.

## Acknowledgements

This proposal was suggested by Dr. Will Klieber.