Uninitialized Reads

ROBERT C. SEACORD, NCC GROUP

UNDERSTANDING THE PROPOSED REVISIONS TO THE C LANGUAGE

Most developers understand that reading uninitialized variables in C is a defect, but some do it anyway—for example, to create entropy. What happens when you read uninitialized objects is unsettled in the current version of the C standard (C11).³ Various proposals have been made to resolve these issues in the planned C2X revision of the standard. Consequently, this is a good time to understand existing behaviors as well as proposed revisions to the standard to influence the evolution of the C language. Given that the behavior of uninitialized reads is unsettled in C11, prudence dictates eliminating uninitialized reads from your code.

This article describes *object initialization*, *indeterminate values*, and *trap representations* and then examines sample programs that illustrate the effects of these concepts on program behavior.

INITIALIZATION

Understanding how and when an object is initialized is necessary to understand the behavior of reading an uninitialized object.

An object whose identifier is declared with no linkage (a file scope object has internal linkage by default) and without the storage-class specifier `static` has *automatic storage duration*. The initial value of the object is *indeterminate*. If an initialization is specified for the object, it is performed each time the declaration or compound literal is reached in the execution of the block; otherwise, the value becomes indeterminate each time the declaration is reached.

Subsection 6.7.9 paragraph 10 of the C11 standard⁴ describes how objects having static or thread storage duration are initialized:

If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.

If an object that has static or thread storage duration is not initialized explicitly, then:

- if it has pointer type, it is initialized to a null pointer;
- if it has arithmetic type, it is initialized to (positive or unsigned) zero;
- if it is an aggregate, every member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;
- if it is a union, the first named member is initialized (recursively) according to these rules, and any padding is initialized to zero bits.

If a stored value of an object has a trap representation and is read by an lvalue expression that does not have character type, the behavior is undefined.

Many of the dynamic allocation functions do not initialize memory. For example, the `malloc` function allocates space for an object whose size is specified by its argument and whose value is indeterminate. For the `realloc` function, any bytes in the new object beyond the size of the old object have indeterminate values.

INDETERMINATE VALUES

In all cases, an uninitialized object has an indeterminate value. The C standard states that an *indeterminate value* can be either an unspecified value or a trap representation. An *unspecified value* is a valid value of the relevant type where the C standard imposes no requirements on which value is chosen in any instance. The phrase “in any instance” is unclear. The word *instance* is defined in English as “a case or occurrence of anything,” but it is unclear from the context what is occurring. The obvious interpretation is that the occurrence is a read.⁹ A *trap representation* is an object representation that need not represent a value of the object type. Note that an unspecified value cannot be a trap representation.

If a stored value of an object has a trap representation and is read by an lvalue expression that does not have character type, the behavior is undefined. Consequently, an automatic variable can be assigned a trap representation without causing undefined behavior, but the value of the variable cannot be read until a proper value is stored in it.

Annex J.2, “Undefined behavior,” summarizes incompletely that behavior is undefined in the following circumstances:

- A trap representation is read by an lvalue expression that does not have character type.
- The value of an object with automatic storage duration is used while it is indeterminate.

The second undefined behavior is much more general [at least with respect to objects with automatic storage duration], because indeterminate values include all unspecified values and trap representations. This (incorrectly) implies that reading an indeterminate value from an object that has allocated, static, or thread storage duration is well-defined behavior unless a trap representation is read by an lvalue expression that does not have character type.

According to the current WG14 Convener, David Keaton, reading an indeterminate value of any storage duration is implicit undefined behavior in C, and the description in Annex J.2 (which is non-normative) is incomplete. This revised definition of the undefined behavior might be stated as “The value of an object is read while it is indeterminate.”

Unfortunately, there is no consensus in the committee or broader community concerning uninitialized reads. Memarian and Sewell conducted a survey among 323 C experts to discover what they believe about the properties that systems software relies on in practice, and what current implementations provide.⁵ The survey gathered the following responses to the question, Is reading an uninitialized variable or `struct` member (with a current mainstream compiler):

- undefined behavior? 139 (43 percent)

- ➔ going to make the result of any expression involving that value unpredictable? 42 (13 percent)
- ➔ going to give an arbitrary and unstable value (maybe with a different value if you read again)? 21 (6 percent)
- ➔ going to give an arbitrary but stable value (with the same value if you read again)? 112 (35 percent)

TRAP REPRESENTATIONS

Trap representation are not always well understood, even by expert C programmers and compiler writers.⁶ A *trap representation* is an object representation that need not represent a value of the object type. Fetching a trap representation *might* perform a trap but is not required to. Performing a trap in C interrupts execution of the program to the extent that no further operations are performed.

Trap representations were introduced into the C language to help in debugging. Uninitialized objects can be assigned a trap representation so that an uninitialized read would trap and consequently be detected by the programmer during development. Some compiler writers would prefer to eliminate trap representations altogether and simply make any uninitialized read undefined behavior—the theory being, why prevent compiler optimizations because of obviously broken code? The counterargument is, why optimize obviously broken code and not simply issue a fatal diagnostic?

Unsigned Integer Types

The C standard states that for unsigned integer types other than `unsigned char`, an object representation is divided into *value bits* and *padding bits* (where padding bits

are optional]. Unsigned integer types use a pure binary representation known as the *value representation*, but the values of any padding bits are unspecified. According to the C standard, some combinations of padding bits might generate trap representations—for example, if one padding bit is a *parity bit*.

A parity bit acts as a check on a set of binary values, calculated in such a way that the number of ones in the set plus the parity bit should always be even (or occasionally, should always be odd). Early computers sometimes required the use of parity RAM, and parity checking could not be disabled. Historically, faulty memory was relatively common, and noticeable parity errors were not uncommon. Since then, errors have become less visible as simple parity RAM has fallen out of use. Errors are now invisible because they are not detected, or they are corrected invisibly with ECC (error-correcting code) RAM. ECC memory can detect and correct the most common kinds of internal data corruption. Modern RAM is believed, with much justification, to be reliable, and error-detecting RAM has largely fallen out of use for noncritical applications. Parity bits and ECC bits are seen by the memory-processing unit but are invisible to the programmer.

No arithmetic operation on known values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types. All other combinations of padding bits are alternative object representations of the value specified by the value bits. Reads of trap representations have undefined behavior. No known current architecture, however, implements trap representations for unsigned

integers of any type stored in memory other than `_Bool`. Consequently, trap representations for most unsigned integer types are an obsolete feature of the C standard.

The `_Bool` Type

The `_Bool` type is a special case of an unsigned type that has an actual memory-representable trap representation on many architectures. Values of type `_Bool` typically occupy one byte. Values in that byte other than 0 or 1 are trap representations. Consequently, an implementation may assume that a byte read of a `_Bool` object produces a value of 0 or 1, and optimize based on that assumption. GCC (GNU Compiler Collection) is an example of an implementation that behaves in this manner.

Because converting any nonzero value to type `_Bool` results in the value 1, type punning is required to create an object of type `_Bool` that contains a determinate bit pattern that does not represent any value of type `_Bool` (and is consequently a trap representation in the current standard).

Undefined behaviors can occur from deductions from which optimizations may follow. Consider the following code, for example:

```
_Bool a, b, c, d, e;  
switch (a|(b<<1)|(c<<2)|(d<<3)|(e<<4))
```

Value range propagation may deduce that the switch argument is in the range 0 to 31 and use that deduction when producing a table jump, so that an arbitrary address is jumped to if one of the values is out of range and, consequently, the switch argument is out of that range.

No existing implementations have been shown to omit the range test for the table jump completely. GCC will optimize out the default case and jump to one of the other cases for an out-of-range argument. Omitting the range test, however, is permitted by the C standard and possibly by an implementation that defines `__STDC_ANALYZABLE__`.

Consider the following code:

```
unsigned char f(unsigned char y) {  
    _Bool a; /* intentionally uninitialized */  
    unsigned char x[2] = { 0, 0 };  
    x[a] = 1;  
}
```

In this example, it is possible that the write to `x[a]` would result in an out-of-bounds store for an implementation that does not define `__STDC_ANALYZABLE__`.

Signed integer types

For signed integer types, the bits of the object representation are divided into three groups: value bits, padding bits, and the sign bit. Padding bits are not necessary; `signed char` in particular cannot have padding bits. If the sign bit is zero, it does not affect the resulting value.

The C standard supports three representations for signed integer values: sign and magnitude, one's complement, and two's complement. An implementation is free to choose which representation to use, although two's complement is the most common. The C standard also states that for sign and magnitude and two's complement, the value with sign bit 1 and all value bits zero can be a trap

representation or a normal value. For one's complement, a value with sign bit 1 and all value bits 1 can be a trap representation or a normal value. In the case of sign and magnitude and one's complement, if this representation is a normal value, it is called a negative zero. For two's complement variables, this is the minimum (most negative) value for the type.

Most two's complement implementations treat all representations as normal values. Likewise, most sign magnitude and one's complement implementations treat negative zero as normal values. The C standards committee was unable to identify any current implementations that treated these representations as trap values, so this is a potentially unused and obsolete feature of the C standard.

Pointer types

An integer may be converted to any pointer type. The result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation. The mapping functions for converting pointers to integers and integers to pointers are intended to be consistent with the addressing structure of the execution environment.

Floating-point types

IEC 60559² requires two kinds of NaNs (not a number): quiet and signaling. The C standards committee has adopted only quiet NaNs. It did not adopt signaling NaNs because it is believed that their utility is too limited for the work required to support them.⁷

The IEC 60559 floating-point standard specifies quiet and signaling NaNs, but these terms can be applied to some non-IEC 60559 implementations as well. For example, the VAX reserved operand and the Cray indefinite are signaling NaNs. In IEC 60559 standard arithmetic, operations that trigger a signaling NaN argument generally return a quiet NaN result, provided no trap is taken. Full support for signaling NaNs implies restartable traps, such as the optional traps specified in the IEC 60559 floating-point standard. The C standard supports the primary utility of quiet NaNs “to handle otherwise intractable situations, such as providing a default value for 0.0/0.0,” as stated in IEC 60559.

Other applications of NaNs may prove useful. Available parts of NaNs have been used to encode auxiliary information—for example, about the origin of the NaN. Signaling NaNs might be candidates for filling uninitialized storage, and their available parts could distinguish uninitialized floating objects. IEC 60559 signaling NaNs and trap handlers potentially provide hooks for maintaining diagnostic information or for implementing special arithmetic.

C support for signaling NaNs, or for auxiliary information that could be encoded in NaNs, is problematic, however. Trap handling varies widely among implementations. Implementation mechanisms may trigger, or fail to trigger, signaling NaNs in mysterious ways. The IEC 60559 floating-point standard recommends that NaNs propagate, but it does not require this, and not all implementations do this. Additionally, the floating-point

standard fails to specify the contents of NaNs through format conversion. Making signaling NaNs predictable imposes optimization restrictions that exceed the anticipated benefits. For these reasons, the C standard neither defines the behavior of signaling NaNs, nor specifies the interpretation of NaN significands.

The x86 Extended Precision Format is an 80-bit format first implemented in the Intel 8087 math coprocessor and is supported by all processors based on the x86 design that incorporate a floating-point unit. Pseudo-infinity, pseudo-zero, pseudo-NaN, unnormal, and pseudo-denormal are all trap representations.

Itanium CPUs, not a thing flag

Itanium CPUs have a NaT (not a thing) flag for each integer register. The NaT flag is used to control speculative execution and may linger in registers that are not properly initialized before use. An 8-bit value may have as many as 257 different values: 0-255 and a NaT value. The C standard, however, explicitly forbids a NaT value for an `unsigned char`. The NaT flag is not a trap representation in C, because a trap representation is an object representation and an object is a region of data storage in the execution environment and not a register flag.⁸

Instead of classifying the Itanium NaT flag as a trap representation, the following language was added to C11 subsection 6.3.2.1 paragraph 2 to account for the possibility of a NaT flag:

If the lvalue designates an object of automatic storage duration that could have been declared with the register storage class (never had its address taken), and that object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined.

This sentence was added to C11 to support the Itanium NaT flag to give compiler developers the latitude to treat applicable uninitialized reads as undefined behavior on all implementations. This undefined behavior applies even to direct reads of objects of type `unsigned char`. The `unsigned char` type normally has a special status in the standard in that values stored in non-bit-field objects may be copied into an object of type `unsigned char [n]` [e.g., by `memcpy`], where `n` is the size of an object of that type.

SAMPLE PROGRAMS

The preceding review of trap representations makes it clear that the `unsigned char` type is the most interesting case. Consider the following code:

```
unsigned char f(unsigned char y) {
    unsigned char x[1]; /* intentionally unit */
    if (x[0] > 10)
        return y/x[0];
    else
        return 10;
}
```

The `unsigned char` array `x` has automatic storage duration and is consequently uninitialized. Because it is declared as an array, the address of `x` is taken, meaning that the read is defined behavior. While the compiler could avoid taking the address, it cannot change the semantics of the code from unspecified value to undefined behavior. Consequently, the compiler is not allowed to translate this code into instructions that might perform a trap. Objects of `unsigned char` type are guaranteed not to have trap values. The read in this example is defined because it is from an object of type `unsigned char` and known to be backed up by memory. It is unclear, however, which value is read and if this value is stable. From this perspective, it could be argued that this behavior is implicitly undefined. Minimally, the standard is unclear and possibly contradictory.

Defect Report #451¹¹ deals with the instability of uninitialized automatic variables. The proposed committee response to this defect report states that any operation performed on indeterminate values will have an indeterminate value as a result. Library functions will exhibit undefined behavior when used on indeterminate values. It is unclear, however, whether `y/x[0]` can result in a trap. Based on the proposed committee response to Defect Report #451, for all types that do not have trap representations, an uninitialized value can appear to change its value, allowing a conforming implementation to print two different values.

Consider the following code:

```
void f(void) {  
    unsigned char x[1]; /* intentionally unit */  
    x[0] ^= x[0];  
    printf("%d\n", x[0]);  
    printf("%d\n", x[0]);  
    return;  
}
```

In this example, the `unsigned char` array `x` is intentionally uninitialized but cannot contain a trap representation because it has a character type. Consequently, the value is both indeterminate and an unspecified value. The bitwise exclusive OR operation, which would produce a zero on an initialized value, will produce an indeterminate result, which may or may not be zero. The two `printf` calls exhibit undefined behavior and, consequently, might do anything, including printing two different values for `x[0]`.

Uninitialized memory has been used as a source of entropy to seed random number generators in OpenSSL, DragonFly BSD, OpenBSD, and elsewhere.¹⁰ If accessing an indeterminate value is undefined behavior, however, compilers may optimize out these expressions, resulting in predictable values.¹

CONCLUSIONS

The behavior associated with uninitialized reads is an unsettled issue that the C Standards Committee needs to address in the next revision of the standard [C2X]. One simple solution would be to eliminate trap representations altogether and simply state that reads of indeterminate

values are undefined behavior. This would greatly simplify the standard (which itself is of value) and provide compiler developers with all the latitude they want to optimize code. The diametrically opposed solution is to define fully concrete semantics for uninitialized reads in which such a read is guaranteed to give the actual contents of memory.

Most likely, some middle ground will be identified that allows compiler optimizations but doesn't eliminate all guarantees for the programmer. One possibility is the introduction of a *wobbly value* that would allow uninitialized objects to change values without requiring this to be undefined behavior.

Trap representations are an oddity, because they were introduced to help diagnose uninitialized reads but are now viewed with suspicion by the safety and security communities, which are wary that the undefined behavior associated with reading a trap value is being imparted to reads of indeterminate values.

References

1. Debian Security Advisory. 2008. DSA-1571-1 openssl—predictable random number generator; <http://www.debian.org/security/2008/dsa-1571>.
2. IEC. 1989. Binary floating-point arithmetic for microprocessor systems [60559:1989].
3. ISO/IEC. 2011. Programming languages—C, 3rd ed. [ISO/IEC 9899:2011]. Geneva, Switzerland.
4. Krebbers, R., Wiedijk, F. N1793: stability of indeterminate values in C11; <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1793.pdf>.
5. Memarian, K., Sewell, P. 2016. Clarifying the C

- memory object model (revised version of WG14 N2012). University of Cambridge; <http://www.cl.cam.ac.uk/~pes20/cerberus/notes64-wg14.html#clarifying-the-c-memory-object-model-uninitialised-values>.
6. Memarian, K., Sewell, P. 2015 (updated 2016). What is C in practice? (Cerberus survey v2): analysis of responses (n2014) - with comments; <https://www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html>.
 7. Open Standards. 2003. Optional support for signaling NaNs; <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1011.htm>.
 8. Peterson, R. 2007. Defect report #338. C99 seems to exclude indeterminate value from being an uninitialized register. Open Standards; http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_338.htm.
 9. Seacord, R. C. 2016. Clarification of unspecified value. Open Standards; <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2042.pdf>.
 10. Wang, X. 2012. More randomness or less; <http://llkqueue.org/blog/2012/06/25/more-randomness-or-less/>.
 11. Wiedijk, F., Krebbers, R. 2013. Defect report #451. Instability of uninitialized automatic variables. Open Standards; http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_451.htm.

Robert C. Seacord is a Principal Security Consultant with NCC Group, where he works with software developers and software development organizations to eliminate vulnerabilities resulting from coding errors before they are

deployed. Robert is the author of six books, including The CERT C Coding Standard, Second Edition [Addison-Wesley, 2014] and Secure Coding in C and C++, Second Edition [Addison-Wesley, 2013]. Robert is on the advisory board for the Linux Foundation and an expert on the ISO/IEC JTC1/SC22/WG14 international standardization working group for the C programming language.

Related Articles

Passing a Language through the Eye of a Needle

- Roberto Ierusalimsky, et al.

How the embeddability of Lua impacted its design

<http://queue.acm.org/detail.cfm?id=1983083>

The Challenge of Cross-language Interoperability

- David Chisnall

Interfacing between languages is increasingly important

<http://queue.acm.org/detail.cfm?id=2543971>

Languages, Levels, Libraries, and Longevity

- John R. Mashey

New programming languages are born every day. Why do some succeed and some fail?

<http://queue.acm.org/detail.cfm?id=1039532>

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.