N1793:
Stability of indeterminate values in C11

Robbert Krebbers and Freek Wiedijk

Radboud University Nijmegen, The Netherlands

Abstract. This paper – document N1793 of WG 14 – proposes and argues for a specific resolution to the issue addressed in Defect Report 451, *Stability of uninitialized variables*.

1 Introduction

This paper focusses on the status according to the C11 standard of *uninitialized variables* with *indeterminate values*, to support the discussion about Defect Report 451. Specifically we address the questions:

- Can uninitialized variables change their value without the program explicitly changing them?
- Exactly when will operations on indeterminate values lead to undefined behavior?

One might think these questions are about the program in Fig. 1, but the issues are more subtle. We will instead look at the program in Fig. 2.

The behavior of this program that we will be arguing for is:

- The C11 standard should be read and/or made more precise as saying that the program in Fig. 2 should print the same number twice. This behavior should hold even if the `i = i` statement is omitted.

```
#include <stdio.h>

int main() {
  int i;
  /* intentionally uninitialized */

  printf("%d\n", i);
  printf("%d\n", i);
  /* does this have to print the same number twice? */
  /* and can this have undefined behavior? */

  return 0;
}
```

Fig. 1.

```
#include <stdio.h>

int main() {
  unsigned char i;
  /* i cannot contain a trap value (6.2.6.1/3 + footnote 49) */

  &i;
  /* i is not in a register (6.3.2.1/2) */

  i = i;
  /* i 'retains its [last-stored] value' (6.2.4/2) from here
     because we 'store' an (albeit indeterminate) value in it */

  printf("%d\n", i);
  printf("%d\n", i);

  return 0;
}
```

Fig. 2.

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

int main() {
  int32_t i;
  &i; i = i;
  printf("%"PRId32"\n", i);
  printf("%"PRId32"\n", i);
  return 0;
}
```

Fig. 3.

– The C11 standard should be read as *not* allowing the program in Fig. 2 to exhibit undefined behavior.

Our main reason for this interpretation is that – ignoring for the moment the decision by the WG 14 committee in DR 260, see Section 2.1 below – there is no clear quote from the C11 standard that allows one to argue *for* indeterminate values changing arbitrarily. A second reason for this interpretation is that it seems desirable that programs like the example program in Section 3.1 below work as intended. One might argue that this interpretation too much restricts what an optimizing compiler can do, but we claim that this is not the case, see the discussion in Section 3.2.

The type `unsigned char` has a special status in the C11 standard, as it is the type of 'bytes' that constitute the object representations. It might seem that our example program somehow relates to this. However, this is not the case: we just used the `unsigned char` type because it is guaranteed not to have trap values

(6.2.6.1/3). It might be replaced by any other type without trap values. For example, in a C11 implementation that has the `int32_t` type, which is another type without trap values, the exact same questions might be asked about the program in Fig. 3.

It might seem academic what happens when one uses uninitialized variables (one should initialize one's variables!), but indeterminate values occur naturally in the padding bytes and bits in structures too. In that case it is not natural to have to initialize these bytes (and it even is difficult to keep them determinate, see 6.2.6.1/6). This means these questions are more than of academic importance. For an example of how uninitialized bytes and bits in structures are related to these issues, see Section 3.1 below.

2 Relevant quotes

2.1 Defect Report 260

Our interpretation of the C11 standard is in conflict with the way the WG 14 committee read the C99 standard in DR 260. Although this Defect Report has been made obsolete by the C11 standard, the decision at that time was that the standard text did *not* need to be changed to get this reading, and the wording of all relevant parts of C99 and C11 are identical.

Here are the relevant quotes from the Defect Report. One of the questions in the Defect Report was:

> *If an object holds an indeterminate value, can that value change other than by an explicit action of the program?*

And the Committee Response was (in the final version, date 2004-09-28):

> *Values may have any bit-pattern that validly represents them and the implementation is free to move between alternate representations (for example, it may normalize pointers, floating-point representations etc.). In the case of an indeterminate value all bit-patterns are valid representations and the actual bit-pattern may change without direct action of the program.*

Interestingly enough, an older draft of the answer to this question (dated 2003-03-06) reads:

> *An object with indeterminate value has a bit pattern representation which remains constant during its lifetime. In general, debuggers are not conforming implementations, though nonetheless valuable. A debugger may allow a user to change the value of a variable between statements.*

Apparently the WG 14 committee changed its mind about this back then, which makes the question whether the standard does need to be clarified about this more salient.

2.2 The C11 standard

For convenience here are some relevant quotes from the C11 standard:

3.19.1:

> **indeterminate value**
> either an unspecified value or a trap representation

3.19.2:

> **unspecified value**
> valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance
>
> NOTE An unspecified value cannot be a trap representation.

3.19.3

> **trap representation**
> an object representation that need not represent a value of the object type

6.2.4/1

> An object has a storage duration that determines its lifetime. [...]

6.2.4/2

> The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address,[33)] and retains its last-stored value throughout its lifetime.[34)] If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.
>
> [...]
>
> 34) In the case of a volatile object, the last store need not be explicit in the program.

6.2.4/5

> An object whose identifier is declared with no linkage and without the storage-class specifier `static` has *automatic storage duration*, [...]

6.2.4/6

> For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. [...]

6.2.6.1/3

> Values stored in unsigned bit-fields and objects of type unsigned char shall be represented using a pure binary notation.[49)]
>
> 49) A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. (Adapted from the *American National Dictionary for Information Processing Systems.*) A byte contains `CHAR_BIT` bits, and the values of type `unsigned char` range from 0 to $2^{\texttt{CHAR\_BIT}} - 1$.

6.2.6.1/6

> When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values. [...]

6.3.2.1/2 (cf. DR 338):

> Except when it is the operand of the `sizeof` operator, the unary `&` operator, the `++` operator, the `--` operator, or the left operand of the `.` operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called *lvalue conversion.* [...]
> 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.

6.7.9/10

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

3 Motivating examples

We now present two programs to shed light on why one might like to interpret the C11 standard as implying that indeterminate values should be allowed to change arbitrarily or not.

3.1 Why indeterminate values should not be allowed to change

First, consider a program that wants to put data from its memory in, say, a blob in an XML file. For this it might like to use a hexadecimal or base64 representation. The data it converts like this very well could consist of C structures, and these have by nature 'indeterminate' bytes in their padding (6.2.6.1/6). An example of such a program is shown in Fig. 4.

```
#include <stdio.h>

/* code to dump memory data to stdout */

void printhexdigit(int d) {
  static int col = 0;
  if (col++ >= 40) { putchar('\n'); col = 1; }
  putchar(d < 10 ? '0' + d : 'A' + d - 10);
}

void dumphex(void *b, size_t n) {
  unsigned char *p = b, *lim = p + n, i;
  while (p < lim) { i = *p++;
    printhexdigit(i>>4); printhexdigit(i&0xf); }
}

/* specialize to a struct */

struct foo { short x1; int x2, x3:31; };

void dumpfoo(struct foo *b, int n) {
  dumphex(b, n*sizeof(struct foo));
}

/* use this on actual data */

int main() {
  int i; struct foo a[10];
  for (i = 0; i < 10; i++) a[i].x1 = a[i].x2 = a[i].x3 = 1;
  dumpfoo(a, 10); putchar('\n');
  return 0;
}
```

Fig. 4.

We claim that this program should not exhibit undefined behavior. But if an uninitialized variable with an indeterminate value could change arbitrarily, this program would not function correctly. When trying to print a hexadecimal digit in `printhexdigit`, the value of `d` would change arbitrarily it is trying to print a padding byte, and the value of `'0' + d` would for example not necessarily be a digit character.

The output of a trial run of this program is:

```
010073B701000000010000800100000001000000
01000000010073B701000000010000800100 5FB7
0100000001000080010 0D1BF0100000001000000
0100D1BF01000000010000000010072B701000000
0100008001000408010000000100008001 00D1BF
0100000001000080010072B70100000001000080
```

The string `010073B70100000001000080` at the start of this output represents the first 12 bytes of the array `a`, which is the first `struct foo`. In this the

```
#include <stdio.h>

int main() {
  unsigned char i, j;
  &j; j = j;

  i = 42; printf("i=%d j=%d, ", i, j);
  i = 43; printf("i=%d j=%d, ", i, j);

  /* i will not be used after this point while
     j only becomes live after this point: so
     is a compiler allowed to have i and j share a memory location? */

  j = 44; printf("j=%d\n", j);
  return 0;
}

/* so is this allowed to print 'i=42 j=42, i=43 j=43, j=44'? */
```

Fig. 5.

bytes `73B7` and the most significant bit of the `8` are padding. We argue that having a program like this show undefined behavior when trying to print these bytes/this bit would not be a good thing. Structures like `foo` are ubiquitous (think for example of IP headers), and having code that touch 'unused bits' in these structures behave in weird ways would be very undesirable.

3.2 Why one might think indeterminate values should be allowed to change

Now, consider the program in Fig. 5. If a compiler would do a liveness analysis on this program, if could recognize that `i` and `j` are never life simultaneously, and decide to use the same memory location for them. But then, `j` would change whenever `i` would be modified, and we would get the behavior of a uninitialized variable changing arbitrarily without explicit action from the program.

However, this is not a very strong argument: the liveness analysis easily could take an lvalue conversion of `j` to mean that the variable is live from there on. In other words, even though this *is* an argument for allowing an uninitialized variable to change arbitrarily, it is not a very strong one.

4 Conclusion

A question is whether the standard text needs to be modified for the issues addressed in DR 451, or whether it can remain as it is. If DR 260 had not existed, we would agree that the resolution proposed in this paper can be accepted without modifying the standard text. However, DR 260 clearly shows that at a certain time the current standard text was taken by the committee to have a very different interpretation. Therefore, we argue that at least it should been

made explicit in the standard text that an indeterminate value *cannot* arbitrarily change without direct action by the program.

A related question is exactly what parts of memory can be changed according to the statement from 6.2.4/2:

> *The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.*

This certainly should not hold for bytes from the object representation of that pointer that have been copied elsewhere.

If the committee decides *not* to follow the proposal from this paper, then a clearer distinction should be made between the notions of *indeterminate value* and *unspecified value*. Currently, if one believes 3.19.1

> *indeterminate value = either an unspecified value or a trap representation*

then for a type that has no trap representations these two notions coincide. If an indeterminate value can change arbitrarily (is taken to be 'unstable'), clearly that should then not be the case.

But it will be clear from this paper that we recommend strongly against going that way, and making such a distinction.