ISO/IEC JTC 1/SC 22/WG14 **N 2159**

v 1

July 6, 2017

Flexible array members may take unspecified values

Defect Report against C11

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

Flexible array members are allowed to overlap with trailing padding bytes of their structure. Thus when a value is stored in another member of such a structure object, the bytes of the object representation that correspond to any padding bytes take unspecified values.

—

Acknowledgment: I was pointed to this problem by dorp*, who also largely contributed to the improvement of this document by providing valuable feedback to several of its versions.*

1. INTRODUCTION

The following code illustrates the problem that originated this DR:

```
1  struct FA {
2    size_t size;
3    unsigned char dummy;
4    unsigned char array[];
5  } *s = malloc(sizeof *s + 1);
6  _Static_assert(offsetof(struct FA, array)+3 == sizeof *s,
7                 "we need 3 array elements inside");
8
9  s->array[0] = 1; s->array[1] = 2;
10 s->array[2] = 3; s->array[3] = 4;
11 /* the padding bytes corresponding to s->array[0..2] take
12    unspecified values */
13 s->size = 4;
14 /* this value is now unspecified */
15 printf("%hhu", s->array[0]);
16 /* this value is well specified */
17 printf("%hhu", s->array[3]);
```

Here, on most architectures **sizeof(struct** FA**)** will be such that there is trailing padding (usually 3 or 7 bytes, here we assume 3) after dummy. C11 6.7.2.1 states:

> *In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more* ***trailing padding*** *than the omission would imply.*

So the trailing part of a structure with a flexible array member is considered to be padding. After an assignment to any other structure member such as size in that snippet, the value of any of these padding bytes is unspecified, 6.2.6.1 p6:

> *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.*

In contrast to the first three elements, s->array[3] is a correctly initialized object of type **unsigned char** and value 4.
If this behavior would be the intended, a structure with a flexible array member that uses some of the padding for the initial segment of the array, could not be used reliably.

2. QUESTIONS TO THE COMMITTEE

To be able to formulate our questions we need another, more complete example:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <wchar.h>

typedef struct wstring wstring;

// On machines with 64 size_t, data[0] fits inside.
struct wstring {
  size_t length;
  int status;
  wchar_t data[];
};
_Static_assert(offsetof(wstring, data[1]) <= sizeof(wstring),
               "element␣data[0]␣must␣be␣inside");

// undefined, but not a constraint violation
wstring empty = { 0, 0, { L'\0', }, };
wstring Z     = { 3, 0, L"ZZZ", };

int main (void) {
  wstring A = { 0 };
  printf("size␣of␣wstring:\t%zu\n",  sizeof A);
  printf("size␣of␣empty:\t%zu\n",    sizeof empty);
  printf("size␣of␣Z:\t%zu\n",        sizeof Z);
  printf("offset␣of␣data:\t%zu\n",   offsetof(wstring, data));
  printf("size␣of␣data[0]:\t%zu\n",  sizeof A.data[0]);
  printf("0:␣values:\t(%d)␣%.1ls\n", A.data[0], A.data);
  A.data[0] = L'A';
  printf("1:␣values:\t(%d)␣%.1ls\n", A.data[0], A.data);
  A.length = 1;
  printf("2:␣values:\t(%d)␣%.1ls\n", A.data[0], A.data);

  wstring *B = malloc(sizeof *B);
  *B = A;
  printf("0:␣values:\t(%d)␣%.1ls\n", B->data[0], B->data);
  B->data[0] = L'B';
  printf("1:␣values:\t(%d)␣%.1ls\n", B->data[0], B->data);
  B->length = 1;
  printf("2:␣values:\t(%d)␣%.1ls\n", B->data[0], B->data);

  wstring *C = calloc(1, sizeof *C);
  printf("0:␣values:\t(%d)␣%.1ls\n", C->data[0], C->data);
  C->data[0] = L'C';
  printf("1:␣values:\t(%d)␣%.1ls\n", C->data[0], C->data);
  C->length = 1;
  printf("2:␣values:\t(%d)␣%.1ls\n", C->data[0], C->data);
}
```

(A) (a) Line 28: `A` has a declared type, so it has the same effective type throughout its whole lifetime. What is the effective type of the object located at address `(void*)A.data`?
(b) Line 28: After initialization, before any other value is assigned to any of its members, does `A.data[0]` have a specified value?
(c) Line 30: After assignment to the first array member, does `A.data[0]` have a specified value?
(d) Line 32: After assignment to member object `A.size`, does `A.data[0]` have a specified value?
(B) (a) Line 36: After structure assignment from a valid structure object, what is the effective type of the object located at address `(void*)B->data`, if any?
(b) Line 36: After structure assignment from a valid structure object, does `B->data[0]` have a specified value?
(c) Line 38: After assignment to member object `B->data[0]`, does `B->data[0]` have a specified value?
(d) Line 40: After assignment to member object `B->size`, does `B->data[0]` have a specified value?
(C) (a) Line 43: After zero initialization via **`calloc`**, what is the effective type of the object located at address `(void*)C->data`, if any?
(b) Line 43: Before any other value is assigned to any of its members, does `C->data[0]` have a specified value?
(c) Line 45: After assignment to the first array member, does `C->data[0]` have a specified value?
(d) Line 47: After assignment to member object `C->size`, does `C->data[0]` have a specified value?
(Z) May an initialization of a structure provide initializers for elements of a flexible array?

3. HOW IT SHOULD BE

Our idea for the answers to the questions can be seen in the following. This is not always supported by text in the C standard (thus this DR). We think that it presents a solution of "minimal surprise" for programmers. Quick experiments with `gcc` and `clang` suggest that our view corresponds to the behavior of these implementations.

(A) (a) Line 28: The effective type of the object located at address `(void*)A.data` should be of type **`wchar_t`** for all elements that fit within the boundary of `A`. Otherwise this effective type could never change afterwards and the flexible array would be useless.
(b) Line 28: `A.data[0]` has value `L'\0'`;
(c) Line 30: `A.data[0]` now has the value `L'A'`.
(d) Line 32: `A.data[0]` keeps the value from before.
(B) (a) Line 36: Through structure assignment, `*B` acquires the effective type `wstring`. The effective type of the object located at address `(void*)B->data` is **`wchar_t`**.
(b) Line 36: `B->data[0]` does not have a specified value, so according to different schools of thought the behavior might be

— undefined (anything can happen),
— eradic (the printed value may change on the fly)
— determinable (the value is determined after its first access).

(c) Line 38: `B->data[0]` has value `L'B'`.
(d) Line 40: `B->data[0]` keeps type and value from before.
(C) (a) Line 43: The object located at address `(void*)C->data` has no effective type.
(b) Line 43: The first access `C->data[0]` reads with an lvalue of type **`wchar_t`**, but does not change the effective type. The value that is read is `L'\0'`.

(c) Line 45: The assignment changes the effective type to **wchar_t**, and C->data[0] has the value L'C'.
(d) Line 47: C->data[0] keeps type and value from before.

(Z) Non normative text suggest that the answer is not to allow initialization of flexible arrays. Nevertheless, we found no normative text that states this explicitly. On the other hand, we found compilers (clang and gcc) that allow initialization under certain circumstances.

To summarize, we think that:

(1) Once an object acquires the effective type wstring, either by declaration, assignment or byte copy (e.g **memcpy**), the effective type of overlaid flexible array elements is the base type of the array. So if an object has effective type wstring, the initial segment of the flexible array has to be taken into account for aliasing analysis.
(2) Structure assignment of wstring may or may not affect elements of the flexible array that are overlaid by the structure. All these flexible array elements have unspecified values after structure assignment.
(3) Assignment to any other member object of wstring does **not** affect any flexible array member elements, including the overlaid members or those members that may lay within a **union** that encloses the **struct**. So after such an assignment, the type and value of the array elements are unchanged.
(4) Other than incomplete arrays that are not structure members, the type of flexible array members cannot be completed through initialization. Nevertheless, there are existing implementations that provide the possibility of initialization as an extension, such that enough memory is reserved for the object to accommodate all initialized members.

4. SUGGESTED TECHNICAL CORRIGENDUM

Add a new paragraph after 6.2.6.1 p7:

> Specific rules apply to structures that contain flexible array members. When a value is assigned to an object of such a structure type, bytes within the structure that correspond to the flexible array member take unspecified values. When a value is assigned to a member of an object of such structure type that is not an element of the flexible array, no byte beyond the offset of the flexible array shall be modified. If such a structure is a member of a union type, possibly recursively, the flexible array is not considered to be padding but expands as far as possible to the end of the union.FN1)
>
> FN1) The bytes of a flexible array member that fall inside the structure or an enclosing union are not considered to be padding. They should not change if another member of the structure is modified. On the other hand, a structure assignment operation may not copy any elements of the flexible array. So the elements of the flexible array of the left operand of an assignment have unspecified values after the operation.

To the end of the same section 6.2.6.1 add a forward reference to "flexible arrays".

Add the following sentence to the end of 6.7.2.1 p3:

> The incomplete array type of a flexible array member is never completed. If an initializer for such a flexible array member is provided the behavior is undefined.

Modify 6.7.2.1 p18:

> As a special case, the last element A of a structure S with more than one named member may have an incomplete array type; this is called a flexible

array member. ~~In most situations, the flexible array member is ignored.~~ ~~In particular,~~ the size of S may be as small ~~the structure is~~ as if the flexible array member were omitted, but the offset X of A in S shall be at most **sizeof**(S) ~~except that it may have more trailing padding than the omission would imply~~. The trailing bytes of S starting at X, if any, shall not overlap with other members of S; they are not padding but the initial bytes of the array member A. In particular, if B is the element type of A, and if there is some strictly positive value N such that X+**sizeof**(B[N]) ≤ **sizeof**(S) holds, A designates an "array of B" with at least N elements.FN2) ~~Also~~ However, when a . (or ->) operator has a left operand that is of type (a pointer to) S ~~a structure with a flexible array member~~ and the right operand is A ~~naming that member~~, it behaves as if that member were replaced with the longest array (with ~~the same~~ element type B) that would not make the structure larger than the object being accessed; the offset of the array shall remain X ~~that of the flexible array member~~, even if this would differ from that of the replacement array. If this array would have no elements, it behaves as if it had one element but the behavior is undefined if any attempt is made to access that element or to generate a pointer one past it.

FN2) So for almost all aspects that determine type or value, the part of A that fits into S (if any) is composed of elements of type B. The only difference is that a special rule for the assignment operator applies such that none of the array elements are copied.

Add a footnote to the end of 6.5.16.1 p3:

If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined. FN3)

FN3) If the common type of the operands of a = operation contains a flexible array member, it is unspecified if the elements of that flexible array member are copied, see 6.2.6.1. Such elements will thus have unspecified values after such an assignment operation.