Proposal for C2Y

WG14 N 3197

Title:	Accessing arrays of character type
Author, affiliation:	Robert C. Seacord, Woven by Toyota, United States rcseacord@gmail.com Martin Uecker, Graz University of Technology, Austria uecker@tugraz.at Jens Gustedt, INRIA and ICube, France jens.gustedt@inria.fr
Date:	2023-12-07
Proposal category:	Feature
Target audience:	Implementers
Abstract:	Allowing arrays of character type to be accessed as other object types.
Prior art:	C23

Accessing arrays of character type

Reply-to: Robert C. Seacord (rcseacord@gmail.com)

Document No: **N 3197**

Reference Document: **N 3149**

Date: 2022-12-08

Proposal to allow arrays of character type to be accessed as other object types.

Change Log

2022-12-07:

- Initial version

Table of Contents

1.0 Problem Description

C11 introduced a simple, forward-compatible mechanism for specifying alignments. The following code snippet uses the alignment specifier to ensure that `good_buff` is properly aligned.

```
struct S {
 int i; double d; char c;
};
 int main(void) {
 alignas(struct S) unsigned char good_buff[sizeof(struct S)];
 struct S *good_s_ptr = (struct S *)good_buff;
```

```
 *good_s_ptr = (struct S){ .d = 43.1 };
}
```

This example has undefined behavior from the underlying object `good_buff` being declared as an array of objects of type `unsigned char` and being accessed through an lvalue of type `struct S`. The cast to `(struct S *)`, like any pointer cast, doesn't change the underlying effective type (6.5, paragraph 6) of the storage.

The following example builds cleanly at high warning levels and returns the correct answer on all evaluated implementations (about a dozen):

```
struct S {
  int i; double d; char c;
};


int main(void) {
  alignas(struct S) unsigned char good_buff[sizeof(struct S)];
  struct S *good_s_ptr = (struct S *)good_buff;
  good_s_ptr->i = 100;
  good_s_ptr->d = 12.7;
  good_s_ptr->c = 'a';
  return good_s_ptr->d;
}
```

Godbolt: https://godbolt.org/z/aGeKc68E3

The effective type rules defined in 6.5 Expressions, paragraph 7 allows an object to have its stored value accessed only by an lvalue expression that has a character type. This proposal effectively allows the inverse operation of allowing an array of a character type to be accessed by an lvalue expression of any type.

It's established practice to use areas of character type for low-level storage management. This paper proposes a way to make such code conforming. This makes accessing correctly aligned and sized arrays declared with a non-atomic character type using lvalues of different types well-defined. Currently, these accesses have undefined behavior, but we have been unable to identify compilers which exploit this UB for aliasing analysis. This asymmetric rule does not fit internal models used by compilers for type-based aliasing analysis, which would decide whether two accesses can alias by

considering the types of the two lvalues used for the access. This behavior is unlikely to change because doing so would likely break existing code.

2.0 Proposed Text

Text in green is added to the working draft. ~~Text in red~~ that has been struck through is removed from the working draft. Two different wordings are provided. The first wording makes minimal changes to the standard. The second wording defines allocation functions as returning an array of objects of type **`unsigned char`** to simplify the conceptual behavior and description of effective type.

Proposed Wording 1

Add the following definition after 3.5, paragraph 2:

byte array
object having either no declared type or an array of objects declared with a byte type

byte type
non-atomic character type

Modify 6.5,paragraph 6:

The effective type of an object declared with a type other than an unqualified, byte type, for an access to its stored value, is the declared type of the object~~, if any~~.[97] If a value is stored into an object having no declared type through an lvalue having a type that is not a ~~non-atomic character~~ byte type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into ~~an object having no declared type~~ a byte array using **`memcpy`** or **`memmove`**, or is copied as an array of ~~character~~ byte type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to ~~an object having no declared type or~~ a byte array, the effective type of the object is simply the type of the lvalue used for the access. [XX)]

xx) The object needs to have valid alignment and size for the effective type to be accessed.

Modify 7.17.2.1 p2, The **atomic_init** generic function

The **atomic_init** generic function initializes the atomic object pointed to by **obj** to the value **value**, while also initializing any additional state that the implementation might need to carry for the atomic object. If the object ~~has no declared type~~ is a byte array, after the call the effective type is the atomic type A.

Proposed Wording 2

Add the following definition after 3.5, paragraph 2:

byte array
an array of objects declared with a byte type

byte type
non-atomic character type

Modify 6.5,paragraph 6:

The effective type of an object declared with a type other than an unqualified, byte type, for an access to its stored value, is the declared type of the object~~, if any~~.[98)] If a value is stored into a byte array ~~an object having no~~ declared through an lvalue having a type that is not a a ~~non-atomic character~~ byte type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into ~~an object having no declared type~~ a byte array using **`memcpy`** or **`memmove`**, or is copied as an array of non-atomic character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object ~~having no~~ declared as a byte array type, the effective type of the object is simply the type of the lvalue used for the access.[XX)]

98) Allocated objects ~~have no~~ behave as if declared as an array of **`unsigned char`** type.

xx) The object needs to have valid alignment and size for the effective type to be accessed.

Modify 7.24.3, Memory management functions, paragraph 1

The order and contiguity of storage allocated by successive calls to the **`aligned_alloc`**, **`calloc`**, **`malloc`**, and **`realloc`** functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and size less than or equal to the size requested. It may then be used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated). The lifetime of an allocated object extends from the allocation until the deallocation. Each such allocation shall yield a pointer to ~~an object~~ the first element (lowest byte address) of an array of objects of type **`unsigned char`** that is disjoint from any other object, converted to **`void*`**. ~~The pointer returned points to the first element (lowest byte address) of.~~ If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned to indicate an error, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.

Modify 7.24.3.2 p2, the **calloc** function.

The **calloc** function allocates space ~~for~~ as an array of objects of type `unsigned char` suitable for use as an array of **nmemb** objects, each of whose size is **size**. The space is initialized to all bits zero.[361)]

Modify 7.17.2.1 p2, The **atomic_init** generic function

The **atomic_init** generic function initializes the atomic object pointed to by **obj** to the value **value**, while also initializing any additional state that the implementation might need to carry for the atomic object. If the object ~~has no declared type~~ is a byte array, after the call the effective type is the atomic type A.

3.0 Acknowledgements

We would like to recognize the following people for their help with this work: Aaron Ballman, Hana Dusíková, Javier Múgica, Carlos Andrés Ramírez Cataño, and Owen Davis.