Proposal for C2Y

WG14 N 3238

Title:	Accessing byte arrays, v3
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 Philipp Klaus Krause philipp@informatik.uni-frankfurt.de
Date:	2024-04-01
Proposal category:	Feature
Target audience:	Implementers
Abstract:	Allowing arrays of non-atomic character type (coined byte arrays) to be accessed as other object types.
Prior art:	C23

Accessing byte arrays, v3

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

Document No: **N 3238**

Reference Document: **N 3220**

Date: 2024-04-01

Proposal to allow arrays of non-atomic character type (coined byte arrays) to be accessed as other object types.

Change Log

2023-12-07:

- Initial version

2024-02-20:

- Removed second, alternative wording
- Rebased wording to the N3220 working draft
- Some wording improvement

2024-04-01:

- Add support for byte arrays contained in struct.
- Do not imply exceptions for atomic character types.

Table of Contents

Proposal for C2Y **1**
WG14 N 3230 **1**
Change Log 2
Table of Contents 2
1 Problem Description **3**
2 Identified difficulty 5
3 Proposed Text 5
4 Acknowledgements 7

1 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

Byte arrays in structures

In the following example, there is also the issue that the object with effective type **`struct B`** is accessed as part of an assignment with effective type **`struct X`**.

```
struct B { float x; float y; };
struct X { int n; char buf[8]; } x, y;

void foo(struct B *b)
{
  memcpy(x.buf, b, sizeof (struct B));
  y = x;
}
```

Unfortunately, for byte arrays in a structure, existing aliasing behavior on some compilers does not allow changing the effective type. In GCC - in contrast to C++ and Clang, we have:

https://godbolt.org/z/TG6r4q8eq

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. Additionally, the existing text cites “character type” where atomic character types were not meant to be included; so the new text moves the item up and changes to “byte type”, instead.

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.

This proposal was reviewed by the committee at the Strasbourg meeting in Jan 2024, with strong consensus to proceed.

2 Identified difficulty

During the Strasbourg meeting, and later in N3324, objections were raised about certain processors common in small embedded systems that have limited capabilities to implement atomic types. In particular, the storage is not uniform, and some storage does not have the capacity to hold objects with lock-free type. Therefore, such storage could not be the target of a simple transfer of the effective type of `atomic_flag` via `memcpy`, for example. This is already a problem today, since the transfer of an `atomic_flag` into allocated storage could be impossible, depending on which memory allocated storage resides in. This paper would add another case where these architectures would have problems with `atomic_flag`.

As of today, the Small Device C Compiler (SDCC), supports multiple such architectures, but only supports atomic_flag on one of them, MCS-51 (the type of memory used for allocated storage can be configured by the user at link time).

N3324 proposes a possible solution to this problem: making support for changing the type to `atomic_flag` optional, while still allowing changes to all other types.

3 **Proposed Text**

Text in green is added to the N3220 working draft. ~~Text in red~~ that has been struck through is removed from the N3220 working draft.

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 that is not a byte array, for an access to its stored value, is the declared type of the object~~, if any~~.[97] If a value is stored into a byte array ~~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 and the type of the lvalue needs to have at least the qualifications of the declared type of the object, if any.

NOTE: Following a byte-level copy operation, the new object created in the targeted byte array requires additional initialization before it can be accessed as an lvalue of the following types: **`fexcept_t`** (7.6.4), **`femode_t`** (7.6.5), **`fenv_t`** (7.6.6), atomic types (7.12.7), **`jmp_buf`** (7.13), **`va_list`** (7.16), **`FILE`** (7.23.3), **`cnd_t`** (7.28.2), and **`mtx_t`** (7.28.3).

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.

To properly support byte arrays contained in structures or unions modify 6.5.1 p7 by moving "a character type" in the list in paragraph 7 one item up so that it is included in the "aforementioned types" in:

7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:85)

— a type compatible with the effective type of the object,

— a qualified version of a type compatible with the effective type of the object,

— the signed or unsigned type compatible with the underlying type of the effective type of the object,

— the signed or unsigned type compatible with a qualified version of the underlying type of the effective type of the object

— a byte type, or

— an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union)~~, or~~

~~— a character type.~~

4 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.