Document Number: N2537
Submitter: Aaron Peter Bachmann
Submission Date: 2020-06-25
Allow memory-reuse via pointer-casting

Summary

Arbitrary pointer-type-casting and creating arbitrary address-aliases via pointer-type-punning is undesirable. It is mostly undefined behavior according to the standards ISO/IEC 9899:1999 ... ISO/IEC 9899:2018. Memory-reuse as for example in an implementation for `malloc()`, `free()`, `...` is highly desirable. The benefits of the effective-type-rules can remain intact and memory-reuse can again become valid and portable for many situations of practical interest by just adjusting the wording of the standard. It will again become possible to implement `malloc()` in standard-conforming C.

Introduction

This proposal complements n2484 - Make pointer type casting useful without negatively impacting performance in order to allow memory-reuse in situations where accesses via different effective types are sequenced. To keep the document self-contained some of the arguments given in n2484 are repeated.
The following paragraphs will

- briefly look back in time to a period when pointer-type-casting was generally acceptable,
- give examples of reasonable use of pointer-type-casting and memory-reuse by different types,
- give examples of existing practice in language processors (compilers),
- very informally outline the proposal,
- further discuss the topic and
- finally try to rephrase the wording in the standard.

Prior work

- When C emerged, there were no strict aliasing-rules.
- The ANSI-C version of the book "The C-Programming language" by Brian Kernighan and Dennis Ritchie gives an example of an implementation of `malloc()`, `free()`,... using pointer-type-casting.
- Many implementations for the functions of the `malloc()`-family are written in C and many widely deployed implementations use pointer-casting for memory-reuse. A short list of some of the more well known implementations where source code is available to the general public: dlmalloc (Doug Lea), ptmalloc (Wolfram Gloger), jemalloc (Jason Evans, Mozilla Foundation, Facebook), TCmalloc (Google), the malloc-implementations in musl (Rich Felker), the Hoard Memory Allocator (Emery Berger), mimalloc (Microsoft)

- Most embedded systems have one or more memory-pool(s), and memory from these pools is used to store different types during the lifetime of the embedded program.
- Even C++ which is a language more strictly typed than C has placement-new for memory-reuse.
- Compilers (GCC, clang, icc, ...) allow to entirely disable the strict aliasing-rules e. g. via `-fno-strict-aliasing`. Some C-compilers assume no strict aliasing either by default or as the only option: pcc, tcc, the original C-compiler by DMR, Microsoft C-compilers, ...
- Major projects - e. g. Linux - heavily rely on `-fno-strict-aliasing`.
- More fine-grained control over aliasing is available in many existing compilers, e. g. via `attribute((__may_alias__))`

Proposed solution

Make object-pointer-casting (casting a pointer-type to a pointer to a different type) valid and well defined, and make the access via this pointer valid and well defined as well, provided the accesses are sequenced and other restrictions (const, alignment, ...) are honored. Only the memory shall be reusable but not the values stored therein. As an exception values written to memory via `char*`, `unsigned char*`, and `signed char*` shall survive (For a rational see chapter Discussion). Thus every use of the memory via a new effective type according to this proposal must start with a write (except when the prior type was a character-type). Instead of assigning an effective type once and forever, we allow the memory to acquire a new effective type on every write, but the contents (values or bit-pattern stored therein) of the memory are then lost. A partial write shall be valid as well (This may be useful when the object is bigger than required.). The content of memory not written to with appropriate type is unspecified. For declared types the type declared must be `char`, `signed char`, or `unsigned char` or array of arbitrary dimension of `char`, `singed char` or `unsigned char` in order to allow an access as distinct type after the cast.

Discussion

- Except as noted in this proposal and n2484, strict aliasing-rules shall remain intact.
- In the absence of pointer-type casts, nothing changes with respect to the current standard. Thus, we have no slowdown.
- Even in the presence of pointer-casts no need to emit additional loads or stores will arise for the compiler, because the casts valid according to this proposal are purposely limited to avoid this and the associated costs.
- If the compiler cannot see the pointer-cast, nothing changes. For example the promise made by a function-prototype must remain valid for a well-defined program.
- The memory, we want to reuse, must have a certain size. Array of `char` is sufficient for this:

  ```
  static unsigned char _Alignas(16) Memory_pool[MEMORY_POOL_SIZE];
  ```

  By restricting the declared type we are allowed to cast from for memory-reuse to `char`, `signed char` and `unsigned char` we have no need to address the problem of accessing subtypes (e. g. allocate 4 chars with an alignment of 4 from an array of 8 byte doubles).

- It is important to allow values written via `char*` to survive so that we can call `memset()` in an implementation of `calloc()` and still have the desired behaviour guaranteed.

- When memory is allocated by means not covered by the C-Standard usually the memory has no declared type: e. g.

```
void *memmap(void *addr, size_t length, int prot, int flags, ind fd, off_t offset);
```

- For reasonable code there is no conflict if we have memory-reuse. The problem arises merely from the wording of the standard. If we look at `calloc()` and `free()` and the application we will see the memory used is either distinct or the access is sequenced naturally:
  - `calloc()`, `free()`,... use the metadata for book-keeping, but the application does not.
  - From the moment `calloc()` returns, the memory is used solely by the application up to the time `free()` is called. Then the application no longer uses the memory and `free()` can reuse it for example to store pointers for a list of free memory there.
  - The same reasoning applies for all custom-memory-allocators.

- Security: The author is not aware of any implications for security.
- The proposal is conceptually simple. Unfortunately the proposed wording changes are a bit convoluted.

Proposed wording changes

The changes are relative to n2478, 6.5 Expressions

7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[94)]

- a type compatible with the effective type of the object,
- a qualified version of a type compatible with the effective type of the object,
- a type that is the signed or unsigned type corresponding to the effective type of the object,
- a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,
- 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~~.~~,
- (this bullet-point is from n2484) a properly aligned pointer of any object type that has been explicitly converted from a valid pointer of the effective object type as specified in 6.3.2.3.7 if the conversion is in a function-scope or a block-scope within a function-scope and if the pointer which underwent the conversion is only used within the scope, or
- any other type provided that following additional conditions are all met:

- The object has either no declared type or the declared type is a character-type.
- All read accesses to the type used are preceded by a write access via the same type or a compatilble type and the accesses are not separated by any access via another type except for accesses via character-type. Compatible qualified types or types with different signedness or aggregate types as mentioned above are valid as well.