ISO/IEC JTC 1/SC 22/WG14 **N2756 v0**

June 7, 2021

Enforce storage stability
proposal for addition to TS 6010

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

1. INTRODUCTION

Unfortunately, there are many different interpretations of the `C` standard to explain behavior (or not) of uninitialized objects. In particular, it is debated endlessly where and when reading of such objects or acting on the resulting values has undefined behavior or where it leads to unspecified results or unspecified control flow. It seems that some optimizing compilers rely on some of these interpretations, but not always consistently, and serious bugs occur with user code because of implicit expectations how specific kinds of uninitialized objects or bytes will be treated.

Many of the widely implemented interpretations don't seem sound, and no such interpretation that would be sound and cover enough grounds to be acceptable for wide parts of the `C` community is yet available. Therefore one of the goals of the *Memory Object Model Study Group* of WG14 is to move towards such a wide-spread understanding and provide an acceptable model in TS 6010 in addition to the provenance model that is already provided by it.

This paper presents a relatively conservative vision of what such a model could be and stems largely from a common understanding of `C`'s handling of "memory" which we refer to as "Concrete Memory Model". It can be based on a context-free reading of `6.2.6`, *Representations of types*:

> *Except for bit-fields, objects* ***are composed*** *of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined.*

That is a model where objects and their byte representation are completely identified, where all representation bytes initially have a fixed but arbitrarily chosen value, and where these *representation-bytes-that-are-the-objects* are the only state on which the abstract machine acts.

2. LIMITS AND EXPECTATIONS FOR THE CONCRETE MEMORY MODEL

For two main reasons the Concrete Memory Model is not appropriate for modern `C`, and both have to do with uninitialized objects of automatic storage duration that stem from definitions without initializer, and where the user code misses to issue a first assignment of a value for the first use. These situations have clear type semantics: because there is a definition the effective type of the object can always be assumed to be the declared type. Also, the objects in questions and their underlying storage instances generally have a small size, because large definitions have to be avoided for stackoverflow problems, anyhow.

The first reason that the Concrete Memory Model is not satisfactory is that modern compilers enable certain optimizations for objects they know to be uninitialized; the slack provided by the lack of defined behavior seems to be important for an efficient reordering of the execution in the abstract machine according to the as-if rule.

But we think that another misfeature of the Concrete Memory Model is more important, namely that in most cases

the explicit use of uninitialized objects is a bug,

and that compilers should have the latitude to react to such bugs at compile time (by not producing an executable) or at run time (by aborting or giving diagnostics). Compiling such erroneous programs to faulty executables and to eventually push a misinterpretation of uninitialized objects down the chain of the control flow is just disastrous in many cases. Consider the two functions `f` and `g`:

```
1  void f(void) {
2    unsigned char x;
3    // undefined because the address of x is never taken
4    return x;
5  }
6  void g(void) {
7    unsigned char x[1];
8    // defined because the address of x is taken?
9    return x[0];
10 }
```

For us, both present the same erroneous control flow, and we think that whenever a compiler is able to detect them, it should at least provide a diagnostic, and if possible not produce an executable. With the current standard, compilers may choke on `f` but produce an executable for `g`. With our proposal, both functions could be treated equally by a compiler and compilation could be aborted.

2.1. Small storage instances

There are some usages of uninitialized representations for which users have a reasonable expectation to "just work". Two of these seem essential to us in the current `C` habitat, namely:

(1) Structure or union types that are only partially initialized must be copyable, in particular when used as function arguments.
(2) It should be possible to use padding bytes transparently in copying, comparison, serialization or hashing of structure or union types.

The first is morally supported by the current text of the standard that guarantees that the presentation of a structure or union type will never be, as a whole, a trap representation. Then, object definitions with scalar type or atomic type do not have partial initialization (other than writing to the representation bytes, see below) nor padding *bytes*, and so their representation can in general be assumed to either be completely initialized (every byte has a known value) or to be uninitialized (no byte has a known value).
The places where programs assemble representations of scalar values by storing bytes should be rare, and we expect them to know what they are doing. The model presented here happens to guarantee a stable value is represented in cases where a subset of bytes remains uninitialized. But this is just a marginal property of the model, and not in the center of our attention.
So the problems that may arise when deceiving these expectations to use the full representation of an object usually concern small to medium sized storage instances that are known to have a structure or union type.

2.2. Type punning

There is a substantial number of C programs that re-interpret types of objects or representation arrays for various reasons. Mostly these are not to assemble specific byte values to wider types (such as composing a floating point value from their components) but they use a temporary reinterpretation of consecutive representations as arrays with the goal to accelerate computations (*e.g* as with vector operations) or data movement.

The underlying questions about type (the "effective type rule") are difficult and are not in the focus of this paper. But we should provide a "no-bad-surprise-guarantee" for legitimate type reinterpretation (*e.g* by union type punning).

(1) Provided that the alignment is feasible and that the wider type has no trap representation, it should be possible to access byte patterns as lvalues of a wider type.
(2) A storage instance that consists of `n` correctly aligned consecutive elements that have the same representation as type `T` can be accessed as an array of type `T[n]`.

For example, there are valid situations where four consecutive values of type **`uint8_t`** that are sufficiently aligned can be accessed as a **`uint32_t`** if these types exist on the platform. The concrete value may depend on particular properties of the implementation (endianess) but once the byte values are writen, the interpretation as **`uint32_t`** should be stable. Similarly, it should be possible to access complex values as two-element arrays of real values and vice versa.

2.3. Large storage instances

At the other end of the spectrum are large storage instances. If they have static or thread storage duration, they are either explicitly or implicitly initialized. So the main concern that is left for our model are such storage instances of

— automatic storage duration, in particular that stem from VLA, and
— allocated storage duration, mostly large chunks of memory that are allocated through **`malloc`** or similar library functions.

Here, `C` intentionally does not provide default initialization features because they are considered to have non-negligible run time cost. Programmers usually are aware that it is their responsibility that such objects or storage instances are consistently initialized.
Another common property of large allocations is that modern architectures do not assemble them once and for all (which would more-or-less be consistent with the Concrete Memory Model) but that they provide them on a *memory page* base when need arises.
So generally programmers should be aware that large storage instances *do not fit* into the Concrete Memory Model and need some special treatment.

3. MODELING PRINCIPLES

We try to hold on to the modeling principles that are described in the following. The extend to which they can (or even should) be guaranteed are certainly a matter of debate.
The first principle is currently only expressed implicitly in several places. First is is guaranteed that objects do not change value unless a value is stored to them, and then, more generally, there a visibility property for side effects (which also could be IO, for example) between different threads. We think it only makes sense that such properties should also hold for side effects that are sequenced within the same thread.

PRINCIPLE 1 (OBSERVATION STABILITY). *Every observable change to the abstract state shall be either induced by a store operation, by an IO operation or other external events.*

Here, a store operations to the state of the abstract machine may be an assignment to an lvalue or a call to **`memcpy`**, **`atomic_store`**, **`fread`**, **`scanf`**, or a similar operation that stores into representation bytes; IO operations are all C library functions in the `<stdio.h>` header; external events are for example real time clock progress or value changes on **`volatile`** qualified device memory. Also some signals may be raised as the result of an external event, for example when an IO condition is met or a timer has expired; other signals such as **`SIGFPE`** are the result of an internal event of the execution that are deterministically reproducible.
The second principle, is clearly addressed in the standard for writes to sub-objects that are not bit-fields and that have at least the granularity of a *storage unit*, that is that

correspond to an element of a scalar type. Here it is guaranteed that such writes will not affect neighboring storage units, unless they are padding bytes of the underlying structure or union.
This exception for padding is motivated by the possibility that on a given platform it might be more efficient to write a wider word than the object itself. The only important property such that other parts of the program don't get confused about the value is that the additional bytes are not otherwise used by the structure. The possibility to change padding is currently not limited to adjacent bytes, any padding byte of a structure or union object could be modified by a store operation to any member. Therefore, the following principle only claims a property for entire storage instances.

PRINCIPLE 2 (MODIFICATION LOCALITY). *A store operation shall have no observable effect to storage outside the storage instance that is operand to the operation.*

As a consequence of these two principles, in general an lvalue conversion or a read operation through **memcpy**, **atomic_load**, **fwrite** or **printf** is not considered a store operation and must not have an effect to the read-from storage instance.
Note the difference in formulation compared to Principle 1. Here, it is only claimed that there should be no effect to *storage*, other parts of the abstract machine that are not modeled as user accessible storage instances could well be affected. For example a store operation could exhaust some resource and thereby change the behavior of the abstract machine.
It is claimed that the knowledge about the fact that some object is uninitialized can be used to optimize control flow in a way that significantly improves performance of some programs. The changes that we are proposing may not be consistent with such an approach to optimization because it will impose that certain of these control flows (if the object is a subobject of a byte-initialized object) become in fact reachable. Therefore we assume the following principle.

PRINCIPLE 3 (IMPLICIT REACHABILITY). *A path in the control flow that can only be reached if a preceding operation has undefined behavior shall not be skipped unless the program explicitly tags it as unreachable.*

Currently the only tools to indicate non-reachability are extensions, such as platform specific attributes or builtins, or explicitly issuing undefined operations. In the following we suppose that a function **unreachable** would be added to C as it already has been proposed to C++, see https://wg21.link/p0627.

```
void unreachable(char const* message); // message optional
```

If a call to this pseudo-function is hit during execution, the behavior is undefined, and the intent is that the compiler may optimize the code aggressively under the assumption that this will not happen. Observe that there is currently no tool that would allow to specify general input restrictions for function interfaces besides array sizes. Such tools (**unreachable**, **attest** and **testify**) will be proposed in a separate paper (N2757), but for this paper here it is important to have in mind that we suppose that the expectation that a particular control flow cannot be reached can easily be expressed.
The principles introduced so far talk about storage in general, not necessarily addressable storage. So they also hold for objects that are never accessed through their representation, *e.g* that have **register** storage class. The next and final principle claims operational stability for accesses through the representation.

PRINCIPLE 4 (STORAGE STABILITY). *Consecutive valid calls to* **memcpy** *and* **memcmp** *without intermediate store shall be consistent.*

This principle is meant to give guarantees for the following type of code:

```
Type Big0 = { some_thing, };
Type Big1;
... in a galaxy far, far away ...
unsigned char const*restrict source = &Big0;
unsigned char*restrict target = &Big1;
... in yet another galaxy ...
memcpy(target, source, sizeof(Type));
if (memcmp(target, source, sizeof(Type))) {
  unreachable();
}
```

That is a **memcmp** operation that follows a **memcpy** (without another intermediate store operation) to the same storage instances should return `0` to indicate that the contents of the representation bytes is the same. As indicated by the call to **unreachable** the whole call to **memcmp** could then even be optimized away.

This principle also extends to atomic objects. Most atomic operations have one operand which is atomic and another which is only considered as a value. For these, Principle 1 gives enough guarantees. But **atomic_compare_exchange_strong** operations (and similar) need more. They operate on three different entities, the atomic, a `desired` value, and a separate non-atomic object who's *representation* is used as input for the operation (for an `expected` value) and as an output for the operation (for the previous value of the atomic).

The semantic of such an operation is that of a **memcmp** and **memcpy** that are fused together in one single operation. That means in particular that two consecutive **atomic_compare_exchange_strong** have the semantics of an **memcpy** operation with the representation of `expected` as a target and a **memcmp** with the same representation as a source. As a consequence we have that two consecutive valid calls to **atomic_compare_exchange_strong** without other intermediate visible store to the atomic and non-atomic objects involved have to be consistent.

The structure of code that this guarantees can be seen in the following.

```
_Atomic(C) precious;   // some previous concrete value assumed
C desired;             // some previous concrete value assumed
C expected = { 0 };
if (!atomic_compare_exchange_strong(&precious, &expected, desired)
    && !atomic_compare_exchange_strong(&precious, &expected, desired)) {
  puts("another thread is disturbing us");
} else {
  puts("we managed to recover the precious representation");
}
```

That is, if the first compare-exchange operation writes into `expected` as-if by **memcpy**, the **memcmp**-alike comparison that is performed by the second should be consistent.

Contrary to the above, this does not mean that the comparison part of the second compare exchange operation can be omitted; in a multi-threaded program the value of `precious` could change between the two calls. Note also that for a type `C` that has padding bits or bytes, the above cannot be condensed into a single **atomic_exchange** operation. Storing the return of such an operation would only reconstitute the original value that was stored in `precious`, not necessarily its representation.

4. IMPACT

For application code that properly initializes all objects that are subsequently accessed, the intent is that the semantics should be exactly the same as without the changes. If the application has hypothetical control flow with lvalue conversions of uninitialized objects but where the underlying storage instances are stable, less optimization opportunities may

be presented by the compiler. To avoid a possible impact on the performance, such control flow paths should explicitly be marked as unreachable.
Implementations with aggressive optimization that explicitly or implicitly use the fact that certain objects or representation bytes are uninitialized but ignore that they are part of larger objects that have been byte-initialized, may produce executables that are not conforming to this specification. Currently we are aware that this concerns most versions of the LLVM/clang compiler suite, but problems arising from this seem to have been addressed recently or at least substantially reduced in impact.

5. PROPOSED CHANGES

At first, we have to replace the current handwaving that determines if an object is initialized or not by a definition. We propose to use the term "value-initialized" for a situation where all subobjects that compose the object have received a value that is consistent with (or determines) the effective type. In contrast to that, "byte-initialized" is a situation where we assume that parts of the object have been written to and that subsequent lvalue conversions assemble a value from the representation.

CHANGE 1. *Add a new paragraph to* `6.2.6` *after* `p7`*:*

> An object is *value-initialized* if any of the following holds:
>
> — It stems from an object definition with an explicit or implicit initializer.
> — It is a function parameter or a value capture.
> — It is not an array type and a pointer to it has been argument to a call of one of the library functions that store into their pointed-to parameter.[FNT0)]
> — It (or an object that contains it) has been the target of an assignment operation.
> — It has an aggregate type and all its elements or members have been value-initialized.
>
> It is *byte-initialized* if it is not atomic and if any of the following holds:
>
> — Its storage instance is allocated with the **`calloc`** library function.
> — It has scalar, structure or union type and at least one representation byte has a stored value.
> — It has structure or union type and at least one of its members has been value-initialized.
> — It is a member of a byte-initialized structure or union.
> — It is the representation byte of a value-initialized or byte-initialized object.
>
> An object is *initialized* if it is value-initialized or byte-initialized, otherwise it is *uninitialized*.
>
> FNT0) Such library functions are for example **`atomic_init`**, **`atomic_store`**, **`atomic_exchange`**, **`cnd_init`**, **`fegetenv`**, **`fegetexceptflag`**, **`feholdexcept`**, **`mtx_init`**, **`setjmp`**, **`strftime`**, **`thrd_create`**, **`timespec_get`**, **`time`**, and **`tss_create`**. Although this property does not hold for general array objects, it still holds for the first element of such an array (such as the first character in a string) if that is the target of the store operation. Thus with the following it then also holds for one-element arrays.

Observe, that according to that definition an array object as a whole that is not a member of a structure can only be considered initialized if a store operation has been issued to all its elements.

CHANGE 2. *Add a new footnote to the end of* `6.2.6 p8`*:*

FNT1) An object that is value-initialized and such that no representation byte has been overwritten by a byte-store operation or by an assignment to a different union member, does not have a trap representation. An object that is only byte-initialized or for which a representation byte has been overwritten as indicated, possibly admits a trap representation only if the type of the object has such representations.

Next, we propose de new term *stable storage instance* to describe storage instances for which the user may assume that they behave well, that is where a read operation (as such) always has defined behavior.

CHANGE 3. *In 6.2.4, define the term* "stable storage instance"*:*

A storage instance observably represents a stable value for a given thread of execution (or is *stable* for short) if at least one of the following conditions holds and the corresponding effects of all store operations are visible for the thread:

(*1*) As a whole it represents an object that has been initialized, see 6.2.6.

(*2*) If at least one representation byte has been initialized[FNT2)] and the macro `__STDC_STORAGE_GRANULARITY__`
— is undefined or expands to zero, or,
— is strictly positive and the size of the storage instance is smaller or equal.

FNT2) This includes a possible initialization of the initial segment of an allocation as it is performed by a call to the **realloc** library function.

Here (2) tries to model the fact that some architectures handle allocations that are smaller than a given size (the *page size*) differently than large allocations consisting of multiple pages. On such platforms, accessing a previously unallocated page might for example trigger a signal which could be observable by the application, and which would per default allocate the missing page. Such a signal would then not be reproduced for a second access to the same page. Thus, such an access would constitute an observable effect on the state of the abstract machine (the behavior changes after the first access) and that would be in violation of Principle 1.

The proposal is that implementations define a constant `__STDC_STORAGE_GRANULARITY__` that defines that threshold, or if undefined or set to `0` indicates that allocations of any size that have witnessed at least one store operation can be assumed to be stable.

CHANGE 4. *Add to* 6.10.8.1 *(Conditional feature macros)*

`__STDC_STORAGE_GRANULARITY__` If non-zero, the minimum size such that any storage instance with size or less that has an initialized representation byte is stable, or if undefined or zero indicating that there is no such size restriction.

Then, we have to specify exactly what we can expect from such stable storage instances.

CHANGE 5. *In 6.2.4, enforce stability of stable storage instances:*

As long as a stable storage instance is not the target of any further write operation, all read operations of a given lvalue that is represented by the storage instance result in the same value[FNT3)] and reads to different lvalues that have overlapping representations by the storage instance result in consistent values.

FNT3) This property not withstanding, an lvalue conversion is undefined if the representation is a trap representation for the type of the lvalue or if the type is a pointer type and the represented pointer value does not correspond to the address of a valid storage instance.

And also we replace the famous "could have been declared with **register**" exception such that it does not depend on a lexical property of the defined object, but on a effective quality of the underlying storage instance.

CHANGE 6. *In 6.3.2 replace the current wording that is restricted to storage instances that never had their address taken to storage instances that are not stable.*

> ~~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.~~
>
> If the lvalue designates an object that is uninitialized the type for the access shall not be atomic and the underlying storage instance shall be stable.

We also add a general indication to the introduction of the library clause that all storage that is touched by a library call behaves as-if it had been value-initialized (for write operations) or lvalue-converted (for read operations).

CHANGE 7. *Append to* `7.1.4 p7` *(as provided by TS 6010)*

> If the address of an object that is not an array is an argument to a library function or macro call, the effect for the object and the corresponding storage instance is as indicated in the respective clause for the function; if the function or macro is described to store a value to the object, the object is henceforth value-initialized; if they read a value from the object the access is an lvalue conversion and all requirements for lvalue conversions[FNT4)] shall be fulfilled.
>
> FNT4) This concerns in particular requirements for initialization, stability, alignment, and the validity of a stored value for the type of the lvalue.

Features as described in N2757 should be added to the C standard and TS 6010, independently.

6. QUESTION FOR THE MEMORY MODEL SG AND EVENTUALLY FOR WG14

QUESTION 1. *Shall the indicated changes be integrated into TS 6010?*

7. ACKNOWLEDGEMENTS

We'd like to thank Martin Uecker for his valuable feedback on preliminary versions of this paper.