Remove ATOMIC VAR INIT

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 N2390

June 28, 2019 v 1

Remove ATOMIC_VAR_INIT
proposal for integration to C2x

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

Because of syntactical problems and because it had been found to be basically useless for the intended use case, the macro `ATOMIC_VAR_INIT` had been marked obsolete in C17. We propose to remove it completely.

1. PROBLEM DESCRIPTION

The macro `ATOMIC_VAR_INIT` is basically useless for the purpose for which it was designed, namely to initialize any atomic type with a constant expression of the appropriate base type. There are problems because it is subject to macro parameter expansion (causing difficulties with compound literals) and with the fact that compile time constants of a particular base type (e.g for structure types) might not even exist.
On the other hand, all implementations seems to cope well with the normal initialization syntax for variables when extending it to atomics. Therefore, the use of `ATOMIC_VAR_INIT` has been made optional in C17 and the macro itself has been declared obsolete.
Because it is basically useless and problematic to use, we propose to remove it from C2x. Since the macro name uses a reserved prefix, implementations may continue to provide the macro if they want to. They do not need to do anything to stay conforming.

2. SUGGESTED CHANGES

The text concerning it (7.17.2.1) can not be completely removed from C2x because it contains normative text that is important for initialization of atomics. Therefore we propose to keep the second part of 7.17.2.1 p2 as the a new introduction to "Initialization", 7.17.2 p1.
We also propose to amend the example(s) that are following there, such that it does not use the macro, and such they clarify under which circumstances no additional initialization is necessary for race-free access (Example 1) or where explicit race-free initialization is still required (Example 2).
In addition, we also propose to add a sub-setence about initialization to the definition of *data race* (5.1.2.4 p35), and to remove the mention of the macro from "Future library directions" (7.31.10).

Appendix: diffmarks for the proposed changes

Following are those pages that contain diffmarks for the proposed changed against C2x. The procedure is not perfect, in particular there may be changes inside code blocks that are not visible.

17 **NOTE 7** The "inter-thread happens before" relation describes arbitrary concatenations of "sequenced before", "synchronizes with", and "dependency-ordered before" relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with "dependency-ordered before" followed by "sequenced before". The reason for this limitation is that a consume operation participating in a "dependency-ordered before" relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of "sequenced before". The reasons for this limitation are (1) to permit "inter-thread happens before" to be transitively closed and (2) the "happens before" relation, defined below, provides for relationships consisting entirely of "sequenced before".

18 An evaluation A *happens before* an evaluation B if A is sequenced before B or A inter-thread happens before B. The implementation shall ensure that no program execution demonstrates a cycle in the "happens before" relation.

19 **NOTE 8** This cycle would otherwise be possible only through the use of consume operations.

20 A *visible side effect* A on an object M with respect to a value computation B of M satisfies the conditions:

— A happens before B, and

— there is no other side effect X to M such that A happens before X and X happens before B.

The value of a non-atomic scalar object M, as determined by evaluation B, shall be the value stored by the visible side effect A.

21 **NOTE 9** If there is ambiguity about which side effect to a non-atomic object is visible, then there is a data race and the behavior is undefined.

22 **NOTE 10** This states that operations on ordinary variables are not visibly reordered. This is not actually detectable without data races, but it is necessary to ensure that data races, as defined here, and with suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution.

23 The value of an atomic object M, as determined by evaluation B, shall be the value stored by some side effect A that modifies M, where B does not happen before A.

24 **NOTE 11** The set of side effects from which a given evaluation might take its value is also restricted by the rest of the rules described here, and in particular, by the coherence requirements below.

25 If an operation A that modifies an atomic object M happens before an operation B that modifies M, then A shall be earlier than B in the modification order of M.

26 **NOTE 12** The requirement above is known as "write-write coherence".

27 If a value computation A of an atomic object M happens before a value computation B of M, and A takes its value from a side effect X on M, then the value computed by B shall either be the value stored by X or the value stored by a side effect Y on M, where Y follows X in the modification order of M.

28 **NOTE 13** The requirement above is known as "read-read coherence".

29 If a value computation A of an atomic object M happens before an operation B on M, then A shall take its value from a side effect X on M, where X precedes B in the modification order of M.

30 **NOTE 14** The requirement above is known as "read-write coherence".

31 If a side effect X on an atomic object M happens before a value computation B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M.

32 **NOTE 15** The requirement above is known as "write-read coherence".

33 **NOTE 16** This effectively disallows compiler reordering of atomic operations to a single object, even if both operations are "relaxed" loads. By doing so, it effectively makes the "cache coherence" guarantee provided by most hardware available to C atomic operations.

34 **NOTE 17** The value observed by a load of an atomic object depends on the "happens before" relation, which in turn depends on the values observed by loads of atomic objects. The intended reading is that there exists an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the "happens before" relation derived as described above, satisfy the resulting constraints as imposed here.

35 The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic or if they access an atomic object that has not been

initialized, and neither happens before the other. Any such data race results in undefined behavior.

36 **NOTE 18** It can be shown that programs that correctly use simple mutexes and `memory_order_seq_cst` operations to prevent all data races, and use no other synchronization operations, behave as though the operations executed by their constituent threads were simply interleaved, with each value computation of an object being the last value stored in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result necessarily has undefined behavior even before such a transformation is applied.

37 **NOTE 19** Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this document, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. Reordering of atomic loads in cases in which the atomics in question might alias is also generally precluded, since this could violate the coherence requirements.

38 **NOTE 20** Transformations that introduce a speculative read of a potentially shared memory location might not preserve the semantics of the program as defined in this document, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection.

5.2 Environmental considerations

5.2.1 Character sets

1 Two sets of characters and their associated collating sequences shall be defined: the set in which source files are written (the *source character set*), and the set interpreted in the execution environment (the *execution character set*). Each set is further divided into a *basic character set*, whose contents are given by this subclause, and a set of zero or more locale-specific members (which are not members of the basic character set) called *extended characters*. The combined set is also called the *extended character set*. The values of the members of the execution character set are implementation-defined.

2 In a character constant or string literal, members of the execution character set shall be represented by corresponding members of the source character set or by *escape sequences* consisting of the backslash `\` followed by one or more characters. A byte with all bits set to 0, called the *null character*, shall exist in the basic execution character set; it is used to terminate a character string.

Both the basic source and basic execution character sets shall have the following members: the 26 *uppercase letters* of the Latin alphabet

3
```
    A  B  C  D  E  F  G  H  I  J  K  L  M
    N  O  P  Q  R  S  T  U  V  W  X  Y  Z
```

the 26 *lowercase letters* of the Latin alphabet

```
    a  b  c  d  e  f  g  h  i  j  k  l  m
    n  o  p  q  r  s  t  u  v  w  x  y  z
```

the 10 decimal *digits*

```
    0  1  2  3  4  5  6  7  8  9
```

the following 29 *graphic characters*

```
    !  "  #  %  &  '  (  )  *  +  ,  -  .  /  :
    ;  <  =  >  ?  [  \  ]  ^  _  {  |  }  ~
```

the space character, and control characters representing horizontal tab, vertical tab, and form feed. The representation of each member of the source and execution basic character sets shall fit in a byte. In both the source and execution basic character sets, the value of each character after `0` in the above list of decimal digits shall be one greater than the value of the previous. In source files, there shall be some way of indicating the end of each line of text; this document treats such an end-of-line indicator as if it were a single new-line character. In the basic execution character set, there shall be control characters representing alert, backspace, carriage return, and new line. If any other characters are encountered in a source file (except in an identifier, a character constant, a string

7.17.2 Initialization

~~Synopsis replace~~

~~**Description**~~

1 ~~The expands to a token sequence suitable for initializing an atomic object of a type that is initialization-compatible with~~ value~~.~~ An atomic object with automatic storage duration that is not explicitly initialized is initially in an indeterminate state; however, the default (zero) initialization for objects with static or thread-local storage duration is guaranteed to produce a valid state.

2 Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race.

3 **EXAMPLE 1** The following definitions ensure valid states for `guide` and `head` regardless if these are found in file scope or block scope. Thus any atomic operation that is performed on them after their initialization has been met is well defined.

```
~~        atomic_int guide = ATOMIC_VAR_INIT(42);~~
        _Atomic int guide = 42;
        static void*_Atomic head;
```

4 **EXAMPLE 2** With the following definition in block scope, concurrent accesses to `cumul` are undefined unless a prior race-free initialization, either by a call to **`atomic_init`**, a store operation or by assignment, has been performed.

```
        _Atomic double cumul;
```

7.17.2.1 The `atomic_init` generic function

Synopsis

1
```
        #include <stdatomic.h>
        void atomic_init(volatile A *obj, C value);
```

Description

2 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.

3 Although this function initializes an atomic object, it does not avoid data races; concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race.

4 If a signal occurs other than as the result of calling the **`abort`** or **`raise`** functions, the behavior is undefined if the signal handler calls the **`atomic_init`** generic function.

Returns

5 The **`atomic_init`** generic function returns no value.

6 **EXAMPLE**

```
        _Atomic int guide;
        atomic_init(&guide, 42);
```

7.17.3 Order and consistency

1 The enumerated type **`memory_order`** specifies the detailed regular (non-atomic) memory synchronization operations as defined in 5.1.2.4 and may provide for operation ordering. Its enumeration constants are as follows:[277)]

```
        memory_order_relaxed
        memory_order_consume
        memory_order_acquire
```

[277)] See "future library directions" (7.31.10).

7.31.10 Atomics `<stdatomic.h>`

1 Macros that begin with `ATOMIC_` and an uppercase letter may be added to the macros defined in the `<stdatomic.h>` header. Typedef names that begin with either `atomic_` or `memory_`, and a lowercase letter may be added to the declarations in the `<stdatomic.h>` header. Enumeration constants that begin with `memory_order_` and a lowercase letter may be added to the definition of the `memory_order` type in the `<stdatomic.h>` header. Function names that begin with `atomic_` and a lowercase letter may be added to the declarations in the `<stdatomic.h>` header.

2 The ~~macro `ATOMIC_VAR_INIT` is an obsolescent feature.~~

~~The~~ possibility that an atomic type name of an atomic integer type defines a different type than the corresponding direct type is an obsolescent feature.

7.31.11 Boolean type and values `<stdbool.h>`

1 The ability to undefine and perhaps then redefine the macros `bool`, `true`, and `false` is an obsolescent feature.

7.31.12 Integer types `<stdint.h>`

1 Typedef names beginning with `int` or `uint` and ending with `_t` may be added to the types defined in the `<stdint.h>` header. Macro names beginning with `INT` or `UINT` and ending with `_MAX`, `_MIN`, `_WIDTH`, or `_C` may be added to the macros defined in the `<stdint.h>` header.

7.31.13 Input/output `<stdio.h>`

1 Lowercase letters may be added to the conversion specifiers and length modifiers in `fprintf` and `fscanf`. Other characters may be used in extensions.

2 The use of `ungetc` on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature.

7.31.14 General utilities `<stdlib.h>`

1 Function names that begin with `str` or `wcs` and a lowercase letter may be added to the declarations in the `<stdlib.h>` header.

2 Invoking `realloc` with a `size` argument equal to zero is an obsolescent feature.

7.31.15 String handling `<string.h>`

1 Function names that begin with `str`, `mem`, or `wcs` and a lowercase letter may be added to the declarations in the `<string.h>` header.

7.31.16 Date and time `<time.h>`

Macros beginning with `TIME_` and an uppercase letter may be added to the macros in the `<time.h>` header.

7.31.17 Threads `<threads.h>`

1 Function names, type names, and enumeration constants that begin with either `cnd_`, `mtx_`, `thrd_`, or `tss_`, and a lowercase letter may be added to the declarations in the `<threads.h>` header.

7.31.18 Extended multibyte and wide character utilities `<wchar.h>`

1 Function names that begin with `wcs` and a lowercase letter may be added to the declarations in the `<wchar.h>` header.

2 Lowercase letters may be added to the conversion specifiers and length modifiers in `fwprintf` and `fwscanf`. Other characters may be used in extensions.

7.31.19 Wide character classification and mapping utilities `<wctype.h>`

1 Function names that begin with `is` or `to` and a lowercase letter may be added to the declarations in the `<wctype.h>` header.