Clean up atomics

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 **N 2329**

December 8, 2018 **v 1**

Clean up atomics
proposal for integration to C2x

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

Whereas its intent is clear, the text in the C standard that concerns atomics has several consistency problems. There are contradictions and the standard vocabulary is not always applied correctly. This paper attempts to solve these consistency problems, and to provide a path to a straighter and easier to use and to implement specification.
Even though individually the identified flaws are simple to repair, all together they form a relatively large batch of changes (spelled out in an appendix).
—
This is a follow-up of N1955 that resulted from many discussions on the WG14 reflector or elsewhere. In particular, it integrates suggestions by Robert Seacord, Richard Smith, Aaron Ballman, and probably many more.

1. PROBLEM DESCRIPTION

C17 has still a lot of problems concerning atomic types and synchronization. In the following sections, I list all the oddities that each by itself seem **simple to repair**.
All of this is not meant at all to be substituted for the discussions about atomics that we had on the reflector, in particular with respect to C++. The changes that would result from these discussions are orthogonal to what is proposed here.

1.1. Memory order of operators

The following sections on arithmetic operators, all specify that if they are applied to an atomic object as an operand of any arithmetic base type, the memory order sematic is `memory_order_seq_cst`:

— 6.2.6.1 Loads and stores of objects with atomic types are done with `memory_order_seq_cst` semantics.
— 6.5.2.4 (postfix `++` and `--`)
— 6.5.16.2 Compound assignment. No constraints formulated concerning traps for integer types. In contrast to that, no floating exceptions for floating types are allowed. Also, this defines atomic operations for all arithmetic operands, including some that don't have library calls (`*=`, `/=`, `%=`, `<<=`, `>>=`)

No such mention is made for

— 6.5.3.1 (prefix `++` and `--`), although it explicitly states that these operators are defined to be equivalent to `+= 1` and `-= 1`, respectively.
— 6.5.16.1 Simple assignment, although the paragraph about store says that such a store should be `memory_order_seq_cst`.

1.2. Integer representations and erroneous operations

Concerning the generic library calls, they state in 7.17.7.5

> *For signed integer types, arithmetic is defined to use two's complement representation with silent wrap-around on overflow; there are no undefined results.*

and

> *For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.*

— Can the sign representation depend on the operation that we apply to an object?
— Are these operations supposed to be consistent between operator and function notation?
— What is an address type?
— What is "no undefined behavior"?
— How is the behavior then defined, when we are not told about it?

1.3. Operators versus generic functions

Then a Note (7.17.7.5 p 5) falsely states

> *The operation of the atomic fetch and modify generic functions are nearly equivalent to the operation of the corresponding* `op=` *compound assignment operators. The only differences are that the compound assignment operators are not guaranteed to operate atomically, ...*

Although there are obviously also operators that act on atomic objects, 5.1.2.4 p 4 gives the completely false impression that atomic operations would only be a matter of the C library:

> *The library defines a number of atomic operations (7.17) ...*

1.4. Synchronizations by library calls and events are not properly identified

In addition to the more descriptive text which parts of the standard are implicated in synchronization, there are also still conceptual errors left to address. In particular, the synchronization properties of cleanup handlers (thread specific storage destructors and **`atexit`** handlers) are not clearly written out. I think that what should be done is relatively clear and I didn't hear of misinterpretations which properties should be guaranteed, but I think that some clarification is in order.

1.5. Pointer types are missing for `atomic_fetch_OP`

In the general introduction (7.17.1 p4) there is explicitly an extension of the notations to atomic pointer types:

> *For atomic pointer types,* `M` *is* **`ptrdiff_t`**.

Whereas the only section where this notation is relevant (7.17.7.5 `atomic_fetch_OP`) is restricted to atomic integer types, but then later talks about the result of such operations on address types.

1.6. Vocabulary

For the vocabulary, there is a mixture of the use of the verb combinations between load/store, read/write and fetch/assign. What is the difference? Is there any?
What is the "strength" of a **`memory_order`** argument?
What are "**`volatile`** as device register" semantics?

1.7. Recommended practices

We recommend to use the weak form of compare exchange whenever such an operation must be done in a loop. But in our examples we use the strong version for that.

1.8. Initialization

The macro **`ATOMIC_VAR_INIT`** has already been declared obsolete in C17. But the text concerning it can not be completely removed from C2x because it contains normative text that is important for initialization of atomics.

1.9. Integer types

If there are atomics at all, atomic versions are required to exist for integer types that optional, namely `[u]intptr_t`.
There seems to be no reason to allow atomic integer types (such as **`atomic_int`**) to be different from the direct types (such as **`_Atomic(int)`**). C17 only claims that their representation and alignment must be the same, and the intent is that they should be exchangeable for functions calls. How that would work for border cases (for users) if the types are not exactly the same is a mystery to me.

1.10. Generic functions

The selection process for the active prototype of generic functions is not clearly specified. In particular it is undefined (by omission) what happens if such a function is called and there is no matching prototype. This can in particular happen for `atomic_compare_exchange` functions that have two pointer arguments that must match to be a valid call.
Since such a situation is easily detectable at compile time, this should not be left undefined but present a constraint violation.

1.11. Facility of using acquire-release consistency on all atomic operations

The default memory order for atomic operations is sequential consistency and therefore the use of atomics with that consistency is straight forward: all arithmetic and bitwise operations are available as compound assignments, and some of them are available as direct generic function calls, without separated `order` argument.
Unfortunately this default consistency has some drawbacks on the user side.

— On many CPU architectures sequential consistency can be orders of magnitude more expensive than acquire-release consistency.
— Other synchronization tools provided by the C library (*e.g.* **`mtx_t`**) only guarantee acquire-release consistency, so when mixing atomics with mutex based locking, sequential consistency only adds costs for no gain.
— Properties of parallel executions are mostly argued with acquire-release consistency because the "happend-before" relation provides a simple thread-local view of causality. The global sequential event ordering that is provided by the default model is mostly unused by programmers.

On the other hand using other consistency models is tedious: we have to use the lengthy generic function forms for operations, if they exist, or we even have to manually synthesize the other operations by mean of **`atomic_compare_exchange_weak`**.

1.12. Atomics and VM types

DR 495 raised the question if there have to be provision to evaluate or not a size expression that is found as part of the operand of a **`_Atomic`** *specifier*. I found no indication that would suggest that this should be treated differently from the general case:

— Such a declaration can appear in exactly the same scopes as VM types that don't use the **`_Atomic`** specifier.
— Unless they are themselves part of **`_Alignof`** or **`sizeof`** expressions these declarations declare a type (either for a **`typedef`** or an object) for which the array size must be known at the point of declaration.

On the other hand, I found the formulation after the patching for C17 a bit confusing, since it is not clear with which priority the rules are applied, nor is it stated clearly that for the general case such size expressions are indeed evaluated. So below I propose a patch that clarifies this situation.

1.13. Atomics and function parameters

In C17, it is not clear from the text how an array parameter with **_Atomic** in the [] is rewritten. It says that it rewrites to the "unqualified" pointer type, but by the terminology this does not include atomics. So what happens to **_Atomic**, here?

1.14. Conclusion

This is

contradictory. (the Note cited above is not normative, but still wrong),
confusing. (= is handled different from op=, operators are not mentioned where they should),
weird. (the sign representation is described as the result of an operation, not as the value representation of a data type; what is "no undefined behavior" of a address operation?)
inconsistent. (operators may result in any sign representation ?, and can trap, generic functions are safe)
incomplete. (the set of operators and generic functions are distinct)
obsolete. (**ATOMIC_VAR_INIT**)
tedious.

2. SUGGESTED CHANGES

The suggested changes are appended as diffmarks in the corresponding pages of the latest working paper. Page numbers are only indications as they change by the simultaneous prensence of the old and new text.
Changes can be categorized in

— editorial changes,
— non-normative changes and
— normative changes.

We will not discuss the editorial changes, here.
All these changes are not intended to change behavior of existing code or even ABI or C – C++ compatibility. In the contrary, much work has been invested to keep things as they are currently.
If there are normative changes these are extensions, such as by adding better synchronization features, or extending the available user interfaces. Some effort has been made to make these user interfaces consistent and thereby to ease programming with C's atomic interface.

2.1. Non-normative changes

The changes that should not change normative aspects are summarized by the following. Headlines are those of the log-entries in the git repository:

2.1.1. streamline env concepts w.r.t atomics. Correctly discuss all clauses that are relevant for synchronization operations in the "data race" clause.

2.1.2. amend lang concepts for atomics

— Summarize the expected properties of implementations that support atomics.
— Summarize the general properties of atomic operations.
— Nail down what "no undefined behavior" probably means for people, that is that these operations are interrupted (in a broad sense) under no circumstances.

2.1.3. amend lang expr for atomics. Add a general text to which operations are read-modify-write operations, to env-concepts and remove the then superfluous text from the specific operator definitions.

2.1.4. amend <stdatomic.h>

— Extend the text about atomic generic functions to pointer types, and remove the now redundant text about properties of operations.
— Correct the infamous "Note" to the real differences from the operator versions and add examples that point out the differences.

2.1.5. remove **atomic_int** *from an example.* We should not encourage the use of the **typedef** atomic integer types anymore but use the specifier version instead.

2.1.6. avoid the singular use of the terms "process" and "communication". "Process" and "communication" are the wrong terminology. Use "execution" and "synchronization", instead.

2.1.7. there is no such thing like a "regular type". Apply the same terminology in the running text for the atomic integer types than in the table.

2.1.8. a new note about the interplay between lock-free and signal. The clause for the lock-free property was almost free of sense.

2.1.9. atomics and **volatile** *qualification.* The note explaining why most of the generic functions have **volatile** qualifications was crude and dubious. Stick to the facts.
NB: The note with the change here is completely removed later by a normative change.

2.1.10. strength of ordering constraints. Explain what the "strength" relation between ordering constraints is.

2.1.11. recommend that atomic integer types are the same as the direct type. The whole concept of atomic integer types is a left over from very early formulations of the concept, before the introduction of the **_Atomic** specifier and the application to general data types. Discourage the use of them and mark them as obsolete.

2.1.12. disambiguate the rules for evaluation of size expressions. Resolve DR 495 by clarifying the rules when size expressions are evaluated or not.

2.1.13. add an example for the ambiguity between **_Atomic** *specifier and qualifier.*

2.2. Normative changes

2.2.1. add synchronization and sequencing of **thrd_exit**. In C17, the synchronization properties of **thrd_exit** are only described indirectly and with some inventive terminology. This patch

— complements the synchronization requirements with respect to **thrd_join**, which were only described, there;
— complements with a synchronization requirements towards the execution of **atexit** and **at_quick_exit** handlers;
— uses the correct terminology to describe unordered execution of destructors.

2.2.2. complement the atomic generic functions with the other scalar operations. There are compound assignment operations that have not been covered with atomic generic functions. Therefore these operations have no easy way to be executed with other than sequential consistency.

2.2.3. introduce modify and fetch generic functions. Compound assignment operations can only be expressed with sequential consistency with exactly the same sematics as the operator (return the result of the operation). Simply add all generic function interfaces that do the operation first and return the result.

2.2.4. widen the scope of the fetch and modify functions to all arithmetic types. There is no reason that floating point types are excluded from generic functions. These might not be as efficient as integer types, but are easy to synthesize (just as other types without hardware atomics). Just do it.

2.2.5. force the prototype of a atomic generic functions. Write down the rules that lead to the choice of a particular prototype of a generic function that is used in a call and enforce these rules.

2.2.6. remove the volatile crap from atomic generic functions. These are *generic* functions anyhow, so there is no reason to impose **volatile**-qualified parameters. The way this was formulated forced implementations to go through a **volatile** access, even though this might not be necessary. Just relax.
(the diffmarks are not very good on this because these **volatile** are inside code.)

2.2.7. also remove volatile from the **atomic_flag** *functions.* Removal of **volatile** from these family of functions is a bit more complicated because they are not yet specified as type generic functions.
On the other hand, if simple **unsigned char** or **int** (or **_Atomic** versions thereof) have sufficient properties to be used as **atomic_flag** it could be more efficient to avoid a volatile access for the operations.
Therefore I propose to make them also type generic functions and leave it to the implementation if they want to have separate functions with and without **volatile**.

2.2.8. lift ambiguity of what has to be happening for **_Atomic** *inside array paramenters.* My interpretation of the current state is that the lack of specification of atomic array parameters is just an overlook. I propose to apply the correct terminology.

3. IMPACT

The proposed changes are such that they should have no immediate impact on user code.

— The set non-normative changes only clarify the specification, by collecting certain properties to more centralized locations, by adding obvious omissions and by removing erroneous non-normative text.
— The normative changes are made such that the only changes imposed are to the implementations, not to users. But even for the implementations, these new interfaces are structured exactly as the existing ones, and present no additional code complexity.
— The changes that remove **volatile** qualifications may simply be ignored by implementations. Implementations would be conforming without them. This is only meant to give leeway to implementations to avoid volatile accesses where this is possible for them.
— The additions concerning synchronization should only formulate the obvious existing practice. *E.g* it should be clear that *thread specific storage* destructors are sequenced with respect to the just finished thread code.
— To ease the transition to a new revision, I also propose a feature test macro that reflects the C standard to which the atomics implementation adheres.

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.

or

```
a = (a + (b + 32765));
```

since the values for `a` and `b` might have been, respectively, 4 and −8 or −17 and 12. However, on a machine in which overflow silently generates some value and where positive and negative overflows cancel, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur.

16 **EXAMPLE 7** The grouping of an expression does not completely determine its evaluation. In the following fragment

```
#include <stdio.h>
int sum;
char *p;
/* ... */
sum = sum * 10 - '0' + (*p++ = getchar());
```

the expression statement is grouped as if it were written as

```
sum = (((sum * 10) - '0') + ((*(p++)) = (getchar())));
```

but the actual increment of `p` can occur at any time between the previous sequence point and the next sequence point (the `;`), and the call to `getchar` can occur at any point prior to the need of its returned value.

Forward references: expressions (6.5), type qualifiers (6.7.3), statements (6.8), floating-point environment <fenv.h> (7.6), the `signal` function (7.14), files (7.21.3).

5.1.2.4 Multi-threaded executions and data races

1 Under a hosted implementation, a program can have more than one *thread of execution* (or *thread*) running concurrently. The execution of each thread proceeds as defined by the remainder of this document. The execution of the entire program consists of an execution of all of its threads.[14)] Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

2 The value of an object visible to a thread T at a particular point is the initial value of the object, a value stored in the object by T, or a value stored in the object by another thread, according to the rules below.

3 **NOTE 1** In some cases, there could instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs.

4 Two expression evaluations *conflict* if one of them modifies a memory location and the other one reads or modifies the same memory location.

5 ~~The library defines~~ There are a number of ~~(7.17) and~~ operations that are specially identified as synchronization operations: if the implementation supports the atomics extension these are operators and generic functions that act on atomic objects (6.5 and 7.17); if the implementation supports the thread extension these are calls to initialization functions (7.26.2), operations on mutexes (~~7.26.4) that are specially identified as synchronization operations.~~ 7.26.3 and 7.26.4), and calls to thread functions (7.26.5). These operations play a special role in making ~~assignments~~ side effects in one thread visible to another. A *synchronization operation* on one or more memory locations is either an *acquire operation*, a *release operation*, both an acquire and release operation, or a *consume operation*. A synchronization operation without an associated memory location is a *fence* and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are *relaxed atomic operations*, which are not synchronization operations but still are indivisible, and atomic *read-modify-write operations*, which ~~have special characteristics.~~ are those operations defined in 6.5 and 7.17 that act on an atomic object by reading its value, by performing an optional operation with that value and by storing back a value into that object.

6 **NOTE 2** For example, a call that acquires a mutex will perform an acquire operation on the locations composing the mutex. Correspondingly, a call that releases the same mutex will perform a release operation on those same locations. Informally, performing a release operation on A forces prior side effects on other memory locations to become visible to other threads

[14)] The execution can usually be viewed as an interleaving of all of the threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving as described below.

that later perform an acquire or consume operation on A. Relaxed atomic operations are not included as synchronization operations although, like synchronization operations, they cannot contribute to data races.

7 All modifications to a particular atomic object M occur in some particular total order, called the *modification order* of M. If A and B are modifications of an atomic object M, and A happens before B, then A shall precede B in the modification order of M, which is defined below.

8 **NOTE 3** This states that the modification orders are expected to respect the "happens before" relation.

9 **NOTE 4** There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads can observe modifications to different variables in inconsistent orders.

10 A *release sequence* headed by a release operation A on an atomic object M is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first operation is A and every subsequent operation either is performed by the same thread that performed the release or is an atomic read-modify-write operation.

11 Certain ~~library calls~~ operations *synchronize with* other ~~library calls~~ operations performed by another thread. In particular, an atomic operation A that performs a release operation on an object M synchronizes with an atomic operation B that performs an acquire operation on M and reads a value written by any side effect in the release sequence headed by A.

12 **NOTE 5** Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation.

13 **NOTE 6** The specifications of the synchronization operations define when one reads the value written by another. For atomic variables, the definition is clear. All operations on a given mutex occur in a single total order. Each mutex acquisition "reads the value written" by the last mutex release.

14 An evaluation A *carries a dependency*[15)] to an evaluation B if:

— the value of A is used as an operand of B, unless:

- B is an invocation of the `kill_dependency` macro,
- A is the left operand of a `&&` or `||` operator,
- A is the left operand of a `?:` operator, or
- A is the left operand of a `,` operator;

or

— A writes a scalar object or bit-field M, B reads from M the value written by A, and A is sequenced before B, or

— for some evaluation X, A carries a dependency to X and X carries a dependency to B.

15 An evaluation A is *dependency-ordered before*[16)] an evaluation B if:

— A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or

— for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.

16 An evaluation A *inter-thread happens before* an evaluation B if A synchronizes with B, A is dependency-ordered before B, or, for some evaluation X:

— A synchronizes with X and X is sequenced before B,

— A is sequenced before X and X inter-thread happens before B, or

[15)] The "carries a dependency" relation is a subset of the "sequenced before" relation, and is similarly strictly intra-thread.
[16)] The "dependency-ordered before" relation is analogous to the "synchronizes with" relation, but uses release/consume in place of release/acquire.

2 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.

3 Values stored in unsigned bit-fields and objects of type **unsigned char** shall be represented using a pure binary notation.[50)]

4 Values stored in non-bit-field objects of any other object type consist of $n \times$ **CHAR_BIT** bits, where n is the size of an object of that type, in bytes. The value may be copied into an object of type **unsigned char** [n] (e.g., by **memcpy**); the resulting set of bytes is called the *object representation* of the value. Values stored in bit-fields consist of m bits, where m is the size specified for the bit-field. The object representation is the set of m bits the bit-field comprises in the addressable storage unit holding it. Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations.

5 Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.[51)] Such a representation is called a trap representation.

6 When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.[52)] The value of a structure or union object is never a trap representation, even though the value of a member of the structure or union object may be a trap representation.

7 When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values.

8 Where an operator is applied to a value that has more than one object representation, which object representation is used shall not affect the value of the result.[53)] Where a value is stored in an object using a type that has more than one object representation for that value, it is unspecified which representation is used, but a trap representation shall not be generated.

9 ~~Loads and stores of objects with atomic types~~ All operations on atomic objects ~~are done with **memory_order_seq_cst** semantics.~~ that do not specify otherwise have **memory_order_seq_cst** memory consistency. If an operation with identical values on the non-atomic type is erroneous,[54)] the atomic operation results in an unspecific object representation, that may or may not be an invalid value for the type, such as an invalid address or a floating point NaN. Thereby such an operation may by itself never raise a signal, a trap, a floating point exception or result otherwise in an interruption of the control flow.[55)]

Forward references: declarations (6.7), expressions (6.5), lvalues, arrays, and function designators (6.3.2.1), order and consistency (7.17.3).

6.2.6.2 Integer types

1 For unsigned integer types other than **unsigned char**, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). If there are N value bits, each bit shall represent a different power of 2 between 1 and 2^{N-1}, so that objects of that type shall be capable of representing values from 0 to $2^N - 1$ using a pure binary representation;

[50)] A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. (Adapted from the *American National Dictionary for Information Processing Systems*.) A byte contains **CHAR_BIT** bits, and the values of type **unsigned char** range from 0 to $2^{\text{CHAR\_BIT}} - 1$.

[51)] Thus, an automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it.

[52)] Thus, for example, structure assignment need not copy any padding bits.

[53)] It is possible for objects `x` and `y` with the same effective type `T` to have the same value when they are accessed as objects of type `T`, but to have different values in other contexts. In particular, if == is defined for type `T`, then `x == y` does not imply that **memcmp**(&x, &y, **sizeof** (T))== 0. Furthermore, `x == y` does not necessarily imply that `x` and `y` have the same value; other operations on values of type `T` might distinguish between them.

[54)] Such erroneous operations may for example incur arithmetic overflow, division by zero or negative shifts.

[55)] Whether or not an atomic operation may be interrupted by a signal depends on the lock-free property of the underlying type.

this shall be known as the value representation. The values of any padding bits are unspecified.[56)]

2 For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. There need not be any padding bits; **signed char** shall not have any padding bits. There shall be exactly one sign bit. Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type (if there are M value bits in the signed type and N in the unsigned type, then $M \leq N$). If the sign bit is zero, it shall not affect the resulting value. If the sign bit is one, the value shall be modified in one of the following ways:

— the corresponding value with sign bit 0 is negated (*sign and magnitude*);

— the sign bit has the value $-(2^M)$ (*two's complement*);

— the sign bit has the value $-(2^M - 1)$ (*ones' complement*).

Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for ones' complement), is a trap representation or a normal value. In the case of sign and magnitude and ones' complement, if this representation is a normal value it is called a *negative zero*.

3 If the implementation supports negative zeros, they shall be generated only by:

— the &, |, ^, ~, <<, and >> operators with operands that produce such a value;

— the +, - , *, /, and % operators where one operand is a negative zero and the result is zero;

— compound assignment operators based on the above cases.

It is unspecified whether these cases actually generate a negative zero or a normal zero, and whether a negative zero becomes a normal zero when stored in an object.

4 If the implementation does not support negative zeros, the behavior of the &, |, ^, ~, <<, and >> operators with operands that would produce such a value is undefined.

5 The values of any padding bits are unspecified.[57)] A valid (non-trap) object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. For any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type.

6 The *precision* of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. The *width* of an integer type is the same but including any sign bit; thus for unsigned integer types the two values are the same, while for signed integer types the width is one greater than the precision.

7 Implementations that support the atomics extension represent all signed integers with two's complement such that the object representation with sign bit 1 and all value bits zero is the minimum value of the type.

6.2.7 Compatible type and composite type

1 Two types have *compatible type* if their types are the same. Additional rules for determining whether two types are compatible are described in 6.7.2 for type specifiers, in 6.7.3 for type qualifiers, and in 6.7.6 for declarators.[58)] Moreover, two structure, union, or enumerated types declared in separate

[56)] Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

[57)] Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

[58)] Two types need not be identical to be compatible.

```
        return p1->m;
}
int g()
{
        union {
                struct t1 s1;
                struct t2 s2;
        } u;
        /* ... */
        return f(&u.s1, &u.s2);
}
```

Forward references: address and indirection operators (6.5.3.2), structure and union specifiers (6.7.2.1).

6.5.2.4 Postfix increment and decrement operators

Constraints

1 The operand of the postfix increment or decrement operator shall have atomic, qualified, or unqualified real or pointer type, and shall be a modifiable lvalue.

Semantics

2 The result of the postfix `++` operator is the value of the operand. As a side effect, the value of the operand object is incremented (that is, the value 1 of the appropriate type is added to it). See the discussions of additive operators and compound assignment for information on constraints, types, and conversions and the effects of operations on pointers. The value computation of the result is sequenced before the side effect of updating the stored value of the operand. With respect to an indeterminately-sequenced function call, the operation of postfix `++` is a single evaluation.~~Postfix `++` on an object with atomic type is a read-modify-write operation with `memory_order_seq_cst` memory order semantics.~~[104)]

3 The postfix `--` operator is analogous to the postfix `++` operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it).

Forward references: additive operators (6.5.6), compound assignment (6.5.16.2).

6.5.2.5 Compound literals

Constraints

1 The type name shall specify a complete object type or an array of unknown size, but not a variable length array type.

2 All the constraints for initializer lists in 6.7.9 also apply to compound literals.

Semantics

3 A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a *compound literal*. It provides an unnamed object whose value is given by the initializer list.[105)]

[104)] Where a pointer to an atomic object can be formed and `E` has integer type, `E++` is equivalent to the following code sequence where *T* is the type of `E`:

```
T *addr = &E;
T old = *addr;
T new;
do {
        new = old + 1;
} while (!atomic_compare_exchange_weak(addr, &old, new));
```

with `old` being the result of the operation.

Special care is necessary if `E` has floating type; see 6.5.16.2.

[105)] Note that this differs from a cast expression. For example, a cast specifies a conversion to scalar types or **`void`** only, and the result of a cast expression is not an lvalue.

```
long l;

l = (c = i);
```

the value of `i` is converted to the type of the assignment expression `c = i`, that is, **`char`** type. The value of the expression enclosed in parentheses is then converted to the type of the outer assignment expression, that is, **`long int`** type.

6 **EXAMPLE 3** Consider the fragment:

```
const char **cpp;
char *p;
const char c = 'A';

cpp = &p;       // constraint violation
*cpp = &c;      // valid
*p = 0;         // valid
```

The first assignment is unsafe because it would allow the following valid code to attempt to change the value of the const object `c`.

6.5.16.2 Compound assignment

Constraints

1 For the operators `+=` and `-=` only, either the left operand shall be an atomic, qualified, or unqualified pointer to a complete object type, and the right shall have integer type; or the left operand shall have atomic, qualified, or unqualified arithmetic type, and the right shall have arithmetic type.

2 For the other operators, the left operand shall have atomic, qualified, or unqualified arithmetic type, and (considering the type the left operand would have after lvalue conversion) each operand shall have arithmetic type consistent with those allowed by the corresponding binary operator.

Semantics

3 A *compound assignment* of the form `E1` *op*`=` `E2` is equivalent to the simple assignment expression `E1 = E1` *op* `(E2)`, except that the lvalue `E1` is evaluated only once, and with respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. ~~If `E1` has an atomic type, compound assignment is a read-modify-write operation with **`memory_order_seq_cst`** memory order semantics.~~

4 **NOTE** Where a pointer to an atomic object can be formed and `E1` and `E2` have integer type, this is equivalent to the following code sequence where *T1* is the type of `E1` and *T2* is the type of `E2`:

```
T1 *addr = &E1;
T2 val = (E2);
T1 old = *addr;
T1 new;
do {
      new = old op val;
} while (!atomic_compare_exchange_weak(addr, &old, new));
```

with `new` being the result of the operation.

If `E1` or `E2` has floating type, then exceptional conditions or floating-point exceptions encountered during discarded evaluations of `new` would also be discarded in order to satisfy the equivalence of `E1` *op*`=` `E2` and `E1 = E1` *op* `(E2)`. For example, if Annex F is in effect, the floating types involved have IEC 60559 formats, and **`FLT_EVAL_METHOD`** is 0, the equivalent code would be:

specify a pair of structures that contain pointers to each other. Note, however, that if `s2` were already declared as a tag in an enclosing scope, the declaration `D1` would refer to *it*, not to the tag `s2` declared in `D2`. To eliminate this context sensitivity, the declaration

```
struct s2;
```

can be inserted ahead of `D1`. This declares a new tag `s2` in the inner scope; the declaration `D2` then completes the specification of the new type.

Forward references: declarators (6.7.6), type definitions (6.7.8).

6.7.2.4 Atomic type specifiers

Syntax

1 *atomic-type-specifier:*

_Atomic (*type-name*)

Constraints

2 Atomic type specifiers shall not be used if the implementation does not support atomic types (see 6.10.8.3).

3 The type name in an atomic type specifier shall not refer to an array type, a function type, an atomic type, or a qualified type.

Semantics

4 The properties associated with atomic types are meaningful only for expressions that are lvalues. If the **_Atomic** keyword is immediately followed by a left parenthesis, it is interpreted as a type specifier (with a type name), not as a type qualifier.

5 **EXAMPLE 1** This disambiguation of the grammar is necessary, because in marginal cases a qualifier may be followed by an opening parenthesis.

```
typedef double toto;

void ic(int const tutu);    // valid prototype, void g(int tutu)
void hc(int const(tutu));   // valid prototype, void g(int tutu)
void gc(int const(toto));   // valid prototype, void g(int(*)(double))

void ia(int _Atomic tutu);  // valid prototype, void g(int tutu)
void ha(int _Atomic(tutu)); // invalid prototype, tutu not a type for _Atomic()
void ga(int _Atomic(toto)); // invalid prototype, two types
```

6.7.3 Type qualifiers

Syntax

1 *type-qualifier:*

const
restrict
volatile
_Atomic

Constraints

2 Types other than pointer types whose referenced type is an object type shall not be restrict-qualified.

3 The **_Atomic** qualifier shall not be used if the implementation does not support atomic types (see 6.10.8.3).

4 The type modified by the **_Atomic** qualifier shall not be an array type or a function type.

the keyword **static** shall appear only in a declaration of a function parameter with an array type, and then only in the outermost array type derivation.

2 If an identifier is declared as having a variably modified type, it shall be an ordinary identifier (as defined in 6.2.3), have no linkage, and have either block scope or function prototype scope. If an identifier is declared to be an object with static or thread storage duration, it shall not have a variable length array type.

Semantics

3 If, in the declaration "`T D1`", `D1` has one of the forms:

`D [` *type-qualifier-list*$_{\text{opt}}$ *assignment-expression*$_{\text{opt}}$ `]`
`D [` *type-qualifier-list*$_{\text{opt}}$ *assignment-expression* `]`
`D [` *type-qualifier-list* **static** *assignment-expression* `]`
`D [` *type-qualifier-list*$_{\text{opt}}$ `*` `]`

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* array of *T*".[147] (See 6.7.6.3 for the meaning of the optional type qualifiers and the keyword **static**.)

4 If the size is not present, the array type is an incomplete type. If the size is * instead of being an expression, the array type is a *variable length array* type of unspecified size, which can only be used in declarations or type names with function prototype scope;[148] such arrays are nonetheless complete types. If the size is an integer constant expression and the element type has a known constant size, the array type is not a variable length array type; otherwise, the array type is a *variable length array* type. (Variable length arrays are a conditional feature that implementations need not support; see 6.10.8.3.)

5 If the size is an expression that is not an integer constant expression: if it occurs in a declaration at function prototype scope, it is treated as if it were replaced by *; otherwise, each time it is evaluated it shall have a value greater than zero. The size of each instance of a variable length array type does not change during its lifetime. Where a size expression is part of the operand of an **_Alignof** operator, that expression is not evaluated. Otherwise, where a size expression is part of the operand of a **sizeof** operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated. ~~Where a size expression is part of the operand of an **_Alignof** operator~~Otherwise, that expression is ~~not evaluated~~evalutated.

6 For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, and are integer constant expressions, then both size specifiers shall have the same constant value. If the two array types are used in a context which requires them to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values.

7 **EXAMPLE 1**

```
float fa[11], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers.

8 **EXAMPLE 2** Note the distinction between the declarations

```
extern int *x;
extern int y[];
```

The first declares `x` to be a pointer to **int**; the second declares `y` to be an array of **int** of unspecified size (an incomplete type), the storage for which is defined elsewhere.

[147] When several "array of" specifications are adjacent, a multidimensional array is declared.
[148] Thus, * can be used only in function declarations that are not definitions (see 6.7.6.3).

or

`D` **(** *identifier-list*$_{\text{opt}}$ **)**

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* function returning the unqualified version of *T*".

6 A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function.

7 A declaration of a parameter as "array of *type*" shall be adjusted to "~~qualified~~ atomic or non-atomic, qualified or unqualified pointer to *type*", where the type qualifiers (if any) are those specified within the `[` and `]` of the array type derivation. If the keyword **`static`** also appears within the `[` and `]` of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

8 A declaration of a parameter as "function returning *type*" shall be adjusted to "pointer to function returning *type*", as in 6.3.2.1.

9 If the list terminates with an ellipsis (`, ...`), no information about the number or types of the parameters after the comma is supplied.[149)]

10 The special case of an unnamed parameter of type **`void`** as the only item in the list specifies that the function has no parameters.

11 If, in a parameter declaration, an identifier can be treated either as a typedef name or as a parameter name, it shall be taken as a typedef name.

12 If the function declarator is not part of a definition of that function, parameters may have incomplete type and may use the `[*]` notation in their sequences of declarator specifiers to specify variable length array types.

13 The storage-class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition.

14 An identifier list declares only the identifiers of the parameters of the function. An empty list in a function declarator that is part of a definition of that function specifies that the function has no parameters. The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.[150)]

15 For two function types to be compatible, both shall specify compatible return types.[151)] Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If one type has a parameter type list and the other type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list, the parameter list shall not have an ellipsis terminator and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions. If one type has a parameter type list and the other type is specified by a function definition that contains a (possibly empty) identifier list, both shall agree in the number of parameters, and the type of each prototype parameter shall be compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier. (In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the adjusted type and each parameter declared with qualified type is taken as having the unqualified version of its declared type.)

149) The macros defined in the `<stdarg.h>` header (7.16) can be used to access arguments that correspond to the ellipsis.
150) See "future language directions" (6.11.6).
151) If both function types are "old style", parameter types are not compared.

3 When a signal occurs and `func` points to a function, it is implementation-defined whether the equivalent of **`signal`**`(sig, SIG_DFL);` is executed or the implementation prevents some implementation-defined set of signals (at least including `sig`) from occurring until the current signal handling has completed; in the case of **`SIGILL`**, the implementation may alternatively define that no action is taken. Then the equivalent of `(*func)(sig);` is executed. If and when the function returns, if the value of `sig` is **`SIGFPE`**, **`SIGILL`**, **`SIGSEGV`**, or any other implementation-defined value corresponding to a computational exception, the behavior is undefined; otherwise the program will resume execution at the point it was interrupted.

4 If the signal occurs as the result of calling the **`abort`** or **`raise`** function, the signal handler shall not call the **`raise`** function.

5 If the signal occurs other than as the result of calling the **`abort`** or **`raise`** function, the behavior is undefined if the signal handler refers to any object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as **`volatile`** **`sig_atomic_t`**, or the signal handler calls any function in the standard library other than

— the **`abort`** function,

— the **`_Exit`** function,

— the **`quick_exit`** function,

— the functions and generic functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free,

— the **`atomic_is_lock_free`** generic function with any atomic argument, or

— the **`signal`** function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. Furthermore, if such a call to the **`signal`** function results in a **`SIG_ERR`** return, the value of **`errno`** is indeterminate.[259)]

6 At program startup, the equivalent of

```
signal(sig, SIG_IGN);
```

may be executed for some signals selected in an implementation-defined manner; the equivalent of

```
signal(sig, SIG_DFL);
```

is executed for all other signals defined by the implementation.

7 Use of this function in a multi-threaded program results in undefined behavior. The implementation shall behave as if no library function calls the **`signal`** function.

Returns

8 If the request can be honored, the **`signal`** function returns the value of `func` for the most recent successful call to **`signal`** for the specified signal `sig`. Otherwise, a value of **`SIG_ERR`** is returned and a positive value is stored in **`errno`**.

Forward references: the **`abort`** function (7.22.4.1), the **`exit`** function (7.22.4.4), the **`_Exit`** function (7.22.4.5), the **`quick_exit`** function (7.22.4.7).

7.14.2 Send signal

7.14.2.1 The `raise` function

Synopsis

1

```
#include <signal.h>
int raise(int sig);
```

[259)] If any signal is generated by an asynchronous signal handler, the behavior is undefined.

7.17 Atomics <stdatomic.h>

7.17.1 Introduction

1 The header <stdatomic.h> defines several macros and declares several types and functions for performing atomic operations on data shared between threads.[261)]

2 Implementations that define the macro **__STDC_NO_ATOMICS__** need not provide this header nor support any of its facilities.

3 The macros defined are

```
__STDC_STDATOMIC_VERSION__
```

which expands to the same token as **__STDC_VERSION__** (*yyyymm*L)[262)] and the *atomic lock-free macros*

```
ATOMIC_BOOL_LOCK_FREE
ATOMIC_CHAR_LOCK_FREE
ATOMIC_CHAR16_T_LOCK_FREE
ATOMIC_CHAR32_T_LOCK_FREE
ATOMIC_WCHAR_T_LOCK_FREE
ATOMIC_SHORT_LOCK_FREE
ATOMIC_INT_LOCK_FREE
ATOMIC_LONG_LOCK_FREE
ATOMIC_LLONG_LOCK_FREE
ATOMIC_POINTER_LOCK_FREE
```

which expand to constant expressions suitable for use in **#if** preprocessing directives and which indicate the lock-free property of the corresponding atomic types (both signed and unsigned); and

```
ATOMIC_FLAG_INIT
```

which expands to an initializer for an object of type **atomic_flag**.

4 The types include

```
memory_order
```

which is an enumerated type whose enumerators identify memory ordering constraints;

```
atomic_flag
```

which is a structure type representing a lock-free, primitive atomic flag; and several atomic analogs of integer types.

5 In the following synopses:

— An *A* refers to an atomic type that is not **const**-qualified.

— A *C* refers to its corresponding non-atomic unqualified type.

— An *M* refers to the type of the other argument for arithmetic operations. For atomic ~~integer~~ arithmetic types, *M* is *C*. For atomic pointer types, *M* is **ptrdiff_t**.

— The functions not ending in **_explicit** have the same semantics as the corresponding **_explicit** function with **memory_order_seq_cst** for the **memory_order** argument.

6 The prototype for a call to a generic function specified in this clause is determined by the pointed-to type *A* of the first argument of the call and the types *C* and *M* are deduced according to the above

[261)] See "future library directions" (7.31.8).

[262)] The intent of this macro is to keep track of the version of this document to which a particular C library implementation of <stdatomic.h> adheres. This is meant to facilitate the transition to a new standard for users, not as a leeway for implementations to delay an upgrade.

rules. Other arguments to the call shall be implicitly convertable to the types that are required by the chosen prototype and are converted accordingly before the call.

7 It is unspecified whether any generic function ~~declared in~~ specified in this clause is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name of a generic function, the behavior is undefined.

~~Many operations are volatile-qualified. The "volatile as device register" semantics have not changed in the standard. This qualification means that volatility is preserved when applying these operations to volatile objects.~~

7.17.2 Initialization

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.~~ ~~See "future library directions" (7.31.8).~~

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 = 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(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:[263)]

```
memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_acq_rel
memory_order_seq_cst
```

2 For **memory_order_relaxed**, no operation orders memory.

3 For **memory_order_release**, **memory_order_acq_rel**, and **memory_order_seq_cst**, a store operation performs a release operation on the affected memory location.

4 For **memory_order_acquire**, **memory_order_acq_rel**, and **memory_order_seq_cst**, a load operation performs an acquire operation on the affected memory location.

5 For **memory_order_consume**, a load operation performs a consume operation on the affected memory location.

6 There shall be a single total order S on all **memory_order_seq_cst** operations, consistent with the "happens before" order and modification orders for all affected locations, such that each **memory_order_seq_cst** operation B that loads a value from an atomic object M observes one of the following values:

— the result of the last modification A of M that precedes B in S, if it exists, or

— if A exists, the result of some modification of M that is not **memory_order_seq_cst** and that does not happen before A, or

— if A does not exist, the result of some modification of M that is not **memory_order_seq_cst**.

7 **NOTE 1** Although it is not explicitly required that S include lock operations, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the "happens before" ordering.

8 **NOTE 2** Atomic operations specifying **memory_order_relaxed** are relaxed only with respect to memory ordering. Implementations still guarantee that any given atomic access to a particular atomic object is indivisible with respect to all other atomic accesses to that object.

9 For an atomic operation B that reads the value of an atomic object M, if there is a **memory_order_seq_cst** fence X sequenced before B, then B observes either the last **memory_order_seq_cst** modification of M preceding X in the total order S or a later modification of M in its modification order.

10 For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a **memory_order_seq_cst** fence X such that A is sequenced before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.

11 For atomic modifications A and B of an atomic object M, B occurs later than A in the modification order of M if:

— there is a **memory_order_seq_cst** fence X such that A is sequenced before X, and X precedes B in S, or

— there is a **memory_order_seq_cst** fence Y such that Y is sequenced before B, and A precedes Y in S, or

— there are **memory_order_seq_cst** fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S.

12 **NOTE 3** The memory orderings of **memory_order** impose different ordering constraints on certain operations. **memory_order_relaxed**, **memory_order_consume**, **memory_order_acquire**, **memory_order_acq_rel** and **memory_order_seq_cst** form an inclusive chain of such constraints, from weakest to strongest. **memory_order_release**

[263)] See "future library directions" (7.31.8).

imposes constraints that are incompatible with `memory_order_consume` and `memory_order_acquire`, and that are stronger than `memory_order_relaxed` and weaker than `memory_order_acq_rel`.

13 Atomic read-modify-write operations shall always read the last value (in the modification order) stored before the write associated with the read-modify-write operation.

14 An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations, such that each evaluation observes the values of variables as computed by the last prior assignment in the sequence. The ordering of evaluations in this sequence shall be such that

— If an evaluation B observes a value computed by A in a different thread, then B does not happen before A.

— If an evaluation A is included in the sequence, then all evaluations that assign to the same variable and happen before A are also included.

15 **NOTE 4** The second requirement disallows "out-of-thin-air", or "speculative" stores of atomics when relaxed atomics are used. Since unordered operations are involved, evaluations can appear in this sequence out of thread order. For example, with `x` and `y` initially zero,

```
// Thread 1:
r1 = atomic_load_explicit(&y, memory_order_relaxed);
atomic_store_explicit(&x, r1, memory_order_relaxed);

// Thread 2:
r2 = atomic_load_explicit(&x, memory_order_relaxed);
atomic_store_explicit(&y, 42, memory_order_relaxed);
```

is allowed to produce `r1 == 42 && r2 == 42`. The sequence of evaluations justifying this consists of:

```
atomic_store_explicit(&y, 42, memory_order_relaxed);
r1 = atomic_load_explicit(&y, memory_order_relaxed);
atomic_store_explicit(&x, r1, memory_order_relaxed);
r2 = atomic_load_explicit(&x, memory_order_relaxed);
```

On the other hand,

```
// Thread 1:
r1 = atomic_load_explicit(&y, memory_order_relaxed);
atomic_store_explicit(&x, r1, memory_order_relaxed);

// Thread 2:
r2 = atomic_load_explicit(&x, memory_order_relaxed);
atomic_store_explicit(&y, r2, memory_order_relaxed);
```

is not allowed to produce `r1 == 42 && r2 == 42`, since there is no sequence of evaluations that results in the computation of 42. In the absence of "relaxed" operations and read-modify-write operations with weaker than `memory_order_acq_rel` ordering, the second requirement has no impact.

Recommended practice

16 The requirements do not forbid `r1 == 42 && r2 == 42` in the following example, with `x` and `y` initially zero:

```
// Thread 1:
r1 = atomic_load_explicit(&x, memory_order_relaxed);
if (r1 == 42)
      atomic_store_explicit(&y, r1, memory_order_relaxed);

// Thread 2:
r2 = atomic_load_explicit(&y, memory_order_relaxed);
if (r2 == 42)
      atomic_store_explicit(&x, 42, memory_order_relaxed);
```

Synopsis

1
```
#include <stdatomic.h>
void atomic_signal_fence(memory_order order);
```

Description

2 Equivalent to `atomic_thread_fence(order)`, except that the resulting ordering constraints are established only between a thread and a signal handler executed in the same thread.

3 **NOTE 1** The `atomic_signal_fence` function can be used to specify the order in which actions performed by the thread become visible to the signal handler.

4 **NOTE 2** Compiler optimizations and reorderings of loads and stores are inhibited in the same way as with `atomic_thread_fence`, but the hardware fence instructions that `atomic_thread_fence` would have inserted are not emitted.

Returns

5 The `atomic_signal_fence` function returns no value.

7.17.5 Lock-free property

1 The atomic lock-free macros indicate the lock-free property of ~~integer and address atomic~~ atomic integer and pointer types. A value of 0 indicates that the type is never lock-free; a value of 1 indicates that the type is sometimes lock-free; a value of 2 indicates that the type is always lock-free.

2 **NOTE 1** In addition to the synchronization properties between threads, the lock-free property of a type warrants that operations are perceived indivisible in the presence of interrupts, see 5.1.2.3.

Recommended practice

3 Operations that are lock-free should also be *address-free*. That is, atomic operations on the same memory location via two different addresses will ~~communicate atomically.~~ synchronize. The implementation should not depend on any ~~per-process~~ execution dependent state. This restriction enables ~~communication via memory mapped into a process~~ synchronization via memory that is mapped into an execution more than once and memory shared between ~~two processes~~concurrent program executions.

7.17.5.1 The `atomic_is_lock_free` generic function

Synopsis

1
```
#include <stdatomic.h>
_Bool atomic_is_lock_free(const A *obj);
```

Description

2 The `atomic_is_lock_free` generic function indicates whether or not atomic operations on objects of the type pointed to by `obj` are lock-free.

Returns

3 The `atomic_is_lock_free` generic function returns nonzero (true) if and only if atomic operations on objects of the type pointed to by the argument are lock-free. In any given program execution, the result of the lock-free query shall be consistent for all pointers of the same type.[264)]

7.17.6 Atomic integer types

1 ~~For~~ If the non-atomic version of the direct type exists, for each line in the following table~~,~~[265)] the atomic type name is declared as a type that has the same representation and alignment requirements as the ~~corresponding~~ direct type.[266)]

[264)] `obj` can be a null pointer.
[265)] See "future library directions" (7.31.8).
[266)] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

Atomic type name	Direct type
atomic_bool	~~_Atomic~~ ~~_Bool~~ _Atomic(_Bool)
atomic_char	~~_Atomic~~ ~~char~~ _Atomic(char)
atomic_schar	~~_Atomic~~ ~~signed char~~ _Atomic(signed char)
atomic_uchar	~~_Atomic~~ ~~unsigned char~~ _Atomic(unsigned char)
atomic_short	~~_Atomic~~ ~~short~~ _Atomic(short)
atomic_ushort	~~_Atomic~~ ~~unsigned short~~ _Atomic(unsigned short)
atomic_int	~~_Atomic~~ ~~int~~ _Atomic(int)
atomic_uint	~~_Atomic~~ ~~unsigned int~~ _Atomic(unsigned int)
atomic_long	~~_Atomic~~ ~~long~~ _Atomic(long)
atomic_ulong	~~_Atomic~~ ~~unsigned long~~ _Atomic(unsigned long)
atomic_llong	~~_Atomic~~ ~~long long~~ _Atomic(long long)
atomic_ullong	~~_Atomic~~ ~~unsigned long long~~ _Atomic(unsigned long long)
atomic_char16_t	~~_Atomic~~ ~~char16_t~~ _Atomic(char16_t)
atomic_char32_t	~~_Atomic~~ ~~char32_t~~ _Atomic(char32_t)
atomic_wchar_t	~~_Atomic~~ ~~wchar_t~~ _Atomic(wchar_t)
atomic_int_least8_t	~~_Atomic~~ ~~int_least8_t~~ _Atomic(int_least8_t)
atomic_uint_least8_t	~~_Atomic~~ ~~uint_least8_t~~ _Atomic(uint_least8_t)
atomic_int_least16_t	~~_Atomic~~ ~~int_least16_t~~ _Atomic(int_least16_t)
atomic_uint_least16_t	~~_Atomic~~ ~~uint_least16_t~~ _Atomic(uint_least16_t)
atomic_int_least32_t	~~_Atomic~~ ~~int_least32_t~~ _Atomic(int_least32_t)
atomic_uint_least32_t	~~_Atomic~~ ~~uint_least32_t~~ _Atomic(uint_least32_t)
atomic_int_least64_t	~~_Atomic~~ ~~int_least64_t~~ _Atomic(int_least64_t)
atomic_uint_least64_t	~~_Atomic~~ ~~uint_least64_t~~ _Atomic(uint_least64_t)
atomic_int_fast8_t	~~_Atomic~~ ~~int_fast8_t~~ _Atomic(int_fast8_t)
atomic_uint_fast8_t	~~_Atomic~~ ~~uint_fast8_t~~ _Atomic(uint_fast8_t)
atomic_int_fast16_t	~~_Atomic~~ ~~int_fast16_t~~ _Atomic(int_fast16_t)
atomic_uint_fast16_t	~~_Atomic~~ ~~uint_fast16_t~~ _Atomic(uint_fast16_t)
atomic_int_fast32_t	~~_Atomic~~ ~~int_fast32_t~~ _Atomic(int_fast32_t)
atomic_uint_fast32_t	~~_Atomic~~ ~~uint_fast32_t~~ _Atomic(uint_fast32_t)
atomic_int_fast64_t	~~_Atomic~~ ~~int_fast64_t~~ _Atomic(int_fast64_t)
atomic_uint_fast64_t	~~_Atomic~~ ~~uint_fast64_t~~ _Atomic(uint_fast64_t)
atomic_intptr_t	~~_Atomic~~ ~~intptr_t~~ _Atomic(intptr_t)
atomic_uintptr_t	~~_Atomic~~ ~~uintptr_t~~ _Atomic(uintptr_t)
atomic_size_t	~~_Atomic~~ ~~size_t~~ _Atomic(size_t)
atomic_ptrdiff_t	~~_Atomic~~ ~~ptrdiff_t~~ _Atomic(ptrdiff_t)
atomic_intmax_t	~~_Atomic~~ ~~intmax_t~~ _Atomic(intmax_t)
atomic_uintmax_t	~~_Atomic~~ ~~uintmax_t~~ _Atomic(uintmax_t)

Recommended practice

2 The representation of an atomic integer type is not required to have the same size as the ~~corresponding regular~~ non-atomic version of the direct type but it should have the same size whenever possible, as it eases effort required to port existing code. It is recommended that the atomic type name defines exactly the corresponding direct type.

7.17.7 Operations on atomic types

1 ~~There are only a few kinds of~~ In addition to the operations on atomic ~~types, though there are many instances of those kinds~~ objects that are described by operators, there are a few kinds of operations that are specified as generic functions. This subclause specifies each ~~general kind~~generic function. After evaluation of its arguments, each of these generic functions forms a single read, write or read-modify-write operation with same general properties as described in 5.1.2.4 and 6.2.6.1.

7.17.7.1 The `atomic_store` generic functions

Synopsis

1
```
#include <stdatomic.h>
void atomic_store(A *object, C desired);
void atomic_store_explicit(A *object, C desired, memory_order order);
```

Description

2 The `order` argument shall not be `memory_order_acquire`, `memory_order_consume`, nor `memory_order_acq_rel`. Atomically replace the value pointed to by `object` with the value of `desired`. Memory is affected according to the value of `order`.

Returns

3 The `atomic_store` generic functions return no value.

7.17.7.2 The `atomic_load` generic functions

Synopsis

1
```
#include <stdatomic.h>
C atomic_load(const A *object);
C atomic_load_explicit(const A *object, memory_order order);
```

Description

2 The `order` argument shall not be `memory_order_release` nor `memory_order_acq_rel`. Memory is affected according to the value of `order`.

Returns

3 Atomically returns the value pointed to by `object`.

7.17.7.3 The `atomic_exchange` generic functions

Synopsis

1
```
#include <stdatomic.h>
C atomic_exchange(A *object, C desired);
C atomic_exchange_explicit(A *object, C desired, memory_order order);
```

Description

2 Atomically replace the value pointed to by `object` with `desired`. Memory is affected according to the value of `order`. These operations are read-modify-write operations (5.1.2.4).

Returns

3 Atomically returns the value pointed to by `object` immediately before the effects.

7.17.7.4 The `atomic_compare_exchange` generic functions

Synopsis

1
```
#include <stdatomic.h>
_Bool atomic_compare_exchange_strong(A *object, C *expected, C desired);
_Bool atomic_compare_exchange_strong_explicit(A *object,
          C *expected, C desired, memory_order success, memory_order failure);
_Bool atomic_compare_exchange_weak(A *object, C *expected, C desired);
_Bool atomic_compare_exchange_weak_explicit(A *object,
           C *expected, C desired, memory_order success, memory_order failure);
```

Description

2 The `failure` argument shall not be `memory_order_release` nor `memory_order_acq_rel`. The `failure` argument shall ~~be no stronger than the success~~ not impose more constraints on the operation than the `success` argument.

3 Atomically, compares the contents of the memory pointed to by `object` for equality with that pointed to by `expected`, and if true, replaces the contents of the memory pointed to by `object` with `desired`, and if false, updates the contents of the memory pointed to by `expected` with that pointed to by `object`. Further, if the comparison is true, memory is affected according to the value of `success`, and if the comparison is false, memory is affected according to the value of `failure`. These operations are atomic read-modify-write operations (5.1.2.4).

4 **NOTE 1** For example, the effect of **`atomic_compare_exchange_strong`** is

```
if (memcmp(object, expected, sizeof (*object)) == 0)
      memcpy(object, &desired, sizeof (*object));
else
      memcpy(expected, object, sizeof (*object));
```

5 A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by `expected` and `object` are equal, it may return zero and store back to `expected` the same memory contents that were originally there.

6 **NOTE 2** This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g. load-locked store-conditional machines.

7 **EXAMPLE** A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop.

```
exp = atomic_load(&cur);
do {
      des = function(exp);
} while (!atomic_compare_exchange_weak(&cur, &exp, des));
```

When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable.

Returns

8 The result of the comparison.

7.17.7.5 The atomic fetch and modify generic functions

1 The following operations perform arithmetic and bitwise computations. ~~All of these~~ These operations are applicable to an object ~~of any atomic integer type . None of these operations is applicable to **`atomic_bool`**~~as long as the non-atomic version of the type can be the left operand of the corresponding `op=` compound assignment.[267] The key, operator, and computation correspondence is:

key	op	computation
add	+	addition
sub	-	subtraction
mult	*	multiplication
div	/	division
or	|	bitwise inclusive or
xor	^	bitwise exclusive or
and	&	bitwise and
lshift	<<	left shift
rshift	>>	right shift

[267] Thus these operations are not permitted for pointers to atomic **`_Bool`**, bitwise operations are not permitted for atomic floating point types, and only "add" and "sub" variants are permitted for atomic pointer types. For the latter the type for *M* is **`ptrdiff_t`**, see 7.17.1.

Synopsis

2
```
#include <stdatomic.h>
C atomic_fetch_key(A *object, M operand);
C atomic_fetch_key_explicit(A *object, M operand, memory_order order);
C atomic_key_fetch(A *object, M operand);
C atomic_key_fetch_explicit(A *object, M operand, memory_order order);
```

Description

3 Atomically replaces the value pointed to by `object` with the result of the computation applied to the value pointed to by `object` and the given operand. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (5.1.2.4). ~~For signed integer types, arithmetic is defined to use two's complement representation with silent wrap-around on overflow; there are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.~~

Returns

4 Atomically, the value pointed to by `object` immediately before the effects ~~.~~ (for **`atomic_fetch_`** *key* variants) or after the effects (for **`atomic_`** *key*`_fetch` variants).

5 **NOTE** ~~The~~ For many aspects the operation of the atomic fetch and modify generic functions are ~~nearly~~ equivalent to the operation of the corresponding *op*= compound assignment operators. ~~The only differences are that the compound assignment operators are not guaranteed to operate atomically, and the value yielded by a compound assignment operator is the updated~~ Notable differences are: according to the variant the value returned by the atomic fetch and modify generic functions is the previous value of the ~~object, whereas the value returned by the is the previous value of~~ atomic object; the memory order can be specified to be less strict than the operator; the possible range of values for `operand` can be narrower or larger than for the operator.

6 **EXAMPLE 1** Provided that the implementation allows such large array sizes, the ~~atomic object.~~ following use of the `+=` operator is valid. Before the generic function call, a conversion of `large` to **`ptrdiff_t`** has to be performed. This may trap before the call or will give an implementation defined result that differs from the operator version.

```
static const size_t large = PTRDIFF_MAX + 1ULL
static unsigned char block[large+1];
static unsigned char*_Atomic cp = block;
cp += large;                  // valid, equivalent to cp = &block[large]
atomic_fetch_add(&cp, large); // invalid
```

7 **EXAMPLE 2** Performing integer promotion instead of a conversion to an unsigned type can have the inverse effect of narrowing the possible range for the compound assignment.

```
_Atomic unsigned char one = 1;
one += INT_MAX;                  // invalid
atomic_fetch_add(&one, INT_MAX); // valid, equivalent to ((one = 0), 1)
```

Here the intermediate value 1+**`INT_MAX`** overflows and either traps or provides an implementation defined value, whereas the argument for the generic function is first converted to **`UCHAR_MAX`**.

7.17.8 Atomic flag type and operations

1 The **`atomic_flag`** type provides the ~~classic~~ test-and-set functionality. It has two states, set and clear.

2 Operations on an object of type **`atomic_flag`** shall be lock free.

3 **NOTE** Hence, as per 7.17.5, the operations should also be address-free. No other type requires lock-free operations, so the **`atomic_flag`** type is the minimum hardware-implemented type needed to conform to this document. The remaining types can be emulated with **`atomic_flag`**, though with less than ideal properties.

4 The macro **`ATOMIC_FLAG_INIT`** may be used to initialize an **`atomic_flag`** to the clear state. An **`atomic_flag`** that is not explicitly initialized with **`ATOMIC_FLAG_INIT`** is initially in an indeterminate state.

5 **EXAMPLE**

```
atomic_flag guard = ATOMIC_FLAG_INIT;
```

6 In the following synopsis *A* denotes **atomic_flag**, possibly **volatile** qualified. It is unspecified if any of the type generic functions has only one variant with **volatile** qualification or also a second variant without.

7.17.8.1 The atomic_flag_test_and_set generic functions

Synopsis

1
```
#include <stdatomic.h>
_Bool atomic_flag_test_and_set(A *object);
_Bool atomic_flag_test_and_set_explicit(A *object, memory_order order);
```

Description

2 Atomically places the atomic flag pointed to by `object` in the set state and returns the value corresponding to the immediately preceding state. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (5.1.2.4).

Returns

3 The **atomic_flag_test_and_set** generic functions return the value that corresponds to the state of the atomic flag immediately before the effects. The return value ~~true~~ **true** corresponds to the set state and the return value ~~false~~ **false** corresponds to the clear state.

7.17.8.2 The atomic_flag_clear generic functions

Synopsis

1
```
#include <stdatomic.h>
void atomic_flag_clear(A *object);
void atomic_flag_clear_explicit(A *object, memory_order order);
```

Description

2 The `order` argument shall not be **memory_order_acquire** nor **memory_order_acq_rel**. Atomically places the atomic flag pointed to by `object` into the clear state. Memory is affected according to the value of `order`.

Returns

3 The **atomic_flag_clear** generic functions return no value.

Returns

4 The **at_quick_exit** function returns zero if the registration succeeds, nonzero if it fails.

Forward references: the **quick_exit** function (7.22.4.7).

7.22.4.4 The exit function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void exit(int status);
```

Description

2 The **exit** function causes normal program termination to occur. No functions registered by the **at_quick_exit** function are called. If a program calls the **exit** function more than once, or calls the **quick_exit** function in addition to the **exit** function, the behavior is undefined.

3 First, all functions registered by the **atexit** function are called, in the reverse order of their registration,[308] except that a function is called after any previously registered functions that had already been called at the time it was registered. If, during the call to any such function, a call to the **longjmp** function is made that would terminate the call to the registered function, the behavior is undefined.

4 A sequence point occurs at the beginning of that procedure and immediately before and immediately after each call to a function registered with **atexit**.

5 Next, all open streams with unwritten buffered data are flushed, all open streams are closed, and all files created by the **tmpfile** function are removed.

6 Finally, control is returned to the host environment. If the value of `status` is zero or **EXIT_SUCCESS**, an implementation-defined form of the status *successful termination* is returned. If the value of `status` is **EXIT_FAILURE**, an implementation-defined form of the status *unsuccessful termination* is returned. Otherwise the status returned is implementation-defined.

Returns

7 The **exit** function cannot return to its caller.

7.22.4.5 The _Exit function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void _Exit(int status);
```

Description

2 The **_Exit** function causes normal program termination to occur and control to be returned to the host environment. No functions registered by the **atexit** function, the **at_quick_exit** function, or signal handlers registered by the **signal** function are called. The status returned to the host environment is determined in the same way as for the **exit** function (7.22.4.4). Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined.

Returns

3 The **_Exit** function cannot return to its caller.

7.22.4.6 The getenv function

Synopsis

1
```
#include <stdlib.h>
char *getenv(const char *name);
```

[308] Each function is called as many times as it was registered, and in the correct order with respect to other registered functions.

Description

2 The **getenv** function searches an *environment list*, provided by the host environment, for a string that matches the string pointed to by `name`. The set of environment names and the method for altering the environment list are implementation-defined. The **getenv** function need not avoid data races with other threads of execution that modify the environment list.[309)]

3 The implementation shall behave as if no library function calls the **getenv** function.

Returns

4 The **getenv** function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **getenv** function. If the specified `name` cannot be found, a null pointer is returned.

7.22.4.7 The `quick_exit` function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void quick_exit(int status);
```

Description

2 The **quick_exit** function causes normal program termination to occur. No functions registered by the **atexit** function or signal handlers registered by the **signal** function are called. If a program calls the **quick_exit** function more than once, or calls the **exit** function in addition to the **quick_exit** function, the behavior is undefined. If a signal is raised while the **quick_exit** function is executing, the behavior is undefined.

3 The **quick_exit** function first calls all functions registered by the **at_quick_exit** function, in the reverse order of their registration,[310)] except that a function is called after any previously registered functions that had already been called at the time it was registered. If, during the call to any such function, a call to the **longjmp** function is made that would terminate the call to the registered function, the behavior is undefined.

4 A sequence point occurs at the beginning of that procedure and immediately before and immediately after each call to a function registered with **at_quick_exit**.

5 Then control is returned to the host environment by means of the function call **_Exit**`(status)`.

Returns

6 The **quick_exit** function cannot return to its caller.

7.22.4.8 The `system` function

Synopsis

1
```
#include <stdlib.h>
int system(const char *string);
```

Description

2 If `string` is a null pointer, the **system** function determines whether the host environment has a *command processor*. If `string` is not a null pointer, the **system** function passes the string pointed to by `string` to that command processor to be executed in a manner which the implementation shall document; this might then cause the program calling **system** to behave in a non-conforming manner or to terminate.

Returns

3 If the argument is a null pointer, the **system** function returns nonzero only if a command processor is available. If the argument is not a null pointer, and the **system** function does return, it returns an

[309)] Many implementations provide non-standard functions that modify the environment list.

[310)] Each function is called as many times as it was registered, and in the correct order with respect to other registered functions.

7.26.5.3 The `thrd_detach` function

Synopsis

1
```
#include <threads.h>
int thrd_detach(thrd_t thr);
```

Description

2 The **`thrd_detach`** function tells the operating system to dispose of any resources allocated to the thread identified by `thr` when that thread terminates. The thread identified by `thr` shall not have been previously detached or joined with another thread.

Returns

3 The **`thrd_detach`** function returns **`thrd_success`** on success or **`thrd_error`** if the request could not be honored.

7.26.5.4 The `thrd_equal` function

Synopsis

1
```
#include <threads.h>
int thrd_equal(thrd_t thr0, thrd_t thr1);
```

Description

2 The **`thrd_equal`** function will determine whether the thread identified by `thr0` refers to the thread identified by `thr1`.

Returns

3 The **`thrd_equal`** function returns zero if the thread `thr0` and the thread `thr1` refer to different threads. Otherwise the **`thrd_equal`** function returns a nonzero value.

7.26.5.5 The `thrd_exit` function

Synopsis

1
```
#include <threads.h>
_Noreturn void thrd_exit(int res);
```

Description

2 For every thread-specific storage key which was created with a non-null destructor and for which the value is non-null, **`thrd_exit`** sets the value associated with the key to a null pointer value and then ~~invokes~~ calls the destructor with its previous value. ~~The order in which destructors are invoked is unspecified.~~ These destructor calls are indeterminately sequenced.

3 If after this process there remain keys with both non-null destructors and values, the implementation repeats this process up to **`TSS_DTOR_ITERATIONS`** times.

4 Following this, the **`thrd_exit`** function terminates execution of the calling thread and sets its result code to `res`. The sequence point at the end of the execution of the **`thrd_exit`** function synchronizes with the completion of a successful call, if any, of the **`thrd_join`** function for the calling thread and with the beginning of all calls of **`atexit`** or **`at_quick_exit`** handlers at program termination.[325)]

5 The program terminates normally after the last thread has been terminated. The behavior is as if the program called the **`exit`** function with the status **`EXIT_SUCCESS`** at thread termination time.

Returns

6 The **`thrd_exit`** function returns no value.

7.26.5.6 The `thrd_join` function

[325)] This leaves it unspecified if threads that are terminated by other means than **`thrd_exit`**, for example by an implementation specific mechanism or because they have not been terminated explicitly before program termination, synchronize with **`atexit`** or **`at_quick_exit`** handlers.

7.31 Future library directions

1 The following names are grouped under individual headers for convenience. All external names described below are reserved no matter what headers are included by the program.

7.31.1 Complex arithmetic <complex.h>

1 The function names

```
cerf            cexpm1          clog2
cerfc           clog10          clgamma
cexp2           clog1p          ctgamma
```

and the same names suffixed with **f** or **l** may be added to the declarations in the <complex.h> header.

7.31.2 Character handling <ctype.h>

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

7.31.3 Errors <errno.h>

1 Macros that begin with **E** and a digit or **E** and an uppercase letter may be added to the macros defined in the <errno.h> header.

7.31.4 Floating-point environment <fenv.h>

1 Macros that begin with **FE_** and an uppercase letter may be added to the macros defined in the <fenv.h> header.

7.31.5 Format conversion of integer types <inttypes.h>

1 Macros that begin with either **PRI** or **SCN**, and either a lowercase letter or **X** may be added to the macros defined in the <inttypes.h> header.

7.31.6 Localization <locale.h>

1 Macros that begin with **LC_** and an uppercase letter may be added to the macros defined in the <locale.h> header.

7.31.7 Signal handling <signal.h>

1 Macros that begin with either **SIG** and an uppercase letter or **SIG_** and an uppercase letter may be added to the macros defined in the <signal.h> header.

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

7.31.9 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.

7.31.10 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**,