Clean up atomics, non-normative changes proposal for integration to C2x

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 **N 2389**

June 7, 2019 **v 1**

Clean up atomics, non-normative changes
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 document is a follow up of the series of documents N1955, N2064, and N2329. It contains an update on the non-normative changes that WG14 has agreed to during the London 2019 meeting.

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 are 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. 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.

1.5. 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.

1.6. 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.

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.
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.
The changes that should not change normative aspects are summarized by the following. Headlines in the following table are those of the log-entries in the git repository. Besides these, there are some purely editorial changes.

title of the patch	clauses	pages
streamline env concepts w.r.t atomics	5.1.2.4 p5 start, p11	*14, 15*
amend lang concepts for atomics	6.2.6.1 p9 6.2.6.2 p7	*39* *40*
amend lang expr for atomics	5.1.2.4 p5 end 6.5.2.4 p2 6.5.16.2 p4, p5	*14* *69* *82, 83*
amend `<stdatomic.h>`	7.17 p1 7.17.5	*247* *249 – 250*
remove **`atomic_int`** from an example	7.17.2.2	*242*
avoid the singular use of the terms "process" and "communication"	7.17.5.3	*246*
there is no such thing like a "regular type"	7.17.6	*246, 247*
a new note about the interplay between lock-free and signal	7.17.5 p2	*246*
atomics and volatile qualification	7.17.1 p7	*241*
strength of ordering constraints	7.17.3 p12 7.17.7.4 p2	*243* *248*
recommend that atomic integer types are the same as the direct type	7.17.6 p2	*247, 385*
add an example for the ambiguity between **`_Atomic`** specifier and qualifier	6.7.2.4 p5	*97*
we recommend using the weak version when compare exchange is used in a loop	6.5.2.4 p2 6.5.16.2 p5	*69* *83*
fix the compound operator examples	6.5.16.2 p5	*83*
a note to make the explicit functions explicit	7.17.7.5 p6, p7	*250*

3. IMPACT

The proposed changes are such that they should have no immediate impact on user code. They are only meant clarify the specification, by collecting certain properties to more centralized locations, by adding obvious omissions and by removing erroneous non-normative text.

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

— A inter-thread happens before X and X inter-thread happens before B.

[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.

32 **EXAMPLE 2** The type designated as "`struct tag (*[5])(float)`" has type "array of pointer to function returning `struct tag`". The array has length five and the function has a single parameter of type `float`. Its type category is array.

Forward references: compatible type and composite type (6.2.7), declarations (6.7).

6.2.6 Representations of types

6.2.6.1 General

1 The representations of all types are unspecified except as stated in this subclause.

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.[53)]

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.[54)] 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.[55)] 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.[56)] 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,[57)] 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, or result otherwise in an interruption of the control flow.[58)]

[53)] 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 $2^{\texttt{CHAR\_BIT}} - 1$.

[54)] 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.

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

[56)] 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.

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

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

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; this shall be known as the value representation. The values of any padding bits are unspecified.[59)]

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.[60)] 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.

[59)] 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.

[60)] 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.

```
struct t2 { int m; };
int f(struct t1 *p1, struct t2 *p2)
{
      if (p1->m  <  0)
            p2->m = -p2->m;
      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.~~[108)]

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.

[108)] Where a pointer to an atomic object can be formed and `E` has integer type or pointer type, `E++` is equivalent to the following code sequence where *A* is the type of `E` and `C` is the corresponding non-atomic, unqualified type:

```
A *addr = &E;
C old = *addr;
C 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.

Semantics

2 In *simple assignment* (=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.

3 If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.

4 **EXAMPLE 1** In the program fragment

```
int f(void);
char c;
/* ... */
if ((c = f()) == -1)
      /* ... */
```

the **int** value returned by the function could be truncated when stored in the **char**, and then converted back to **int** width prior to the comparison. In an implementation in which "plain" **char** has the same range of values as **unsigned char** (and **char** is narrower than **int**), the result of the conversion cannot be negative, so the operands of the comparison can never compare equal. Therefore, for full portability, the variable c would be declared as **int**.

5 **EXAMPLE 2** In the fragment:

```
char c;
int i;
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.

3 If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

Semantics

4 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.~~

5 **NOTE** Where a pointer to an atomic object can be formed and `E1` and `E2` have integer or pointer type, this is ~~equivalent~~ similar to the following code sequence where ~~*T1is the* *A1*~~ is the type of `E1`, *C1* is the corresponding non-atomic and unqualified type of `E1` ~~and~~ , and ~~*T2*~~*C2* is the non-atomic and unqualifid type of `E2`:

```
~~~~~~T1 *addr = &E1;~~
~~~~~~T2 val = (E2);~~
~~~~~~T1 old = *addr;~~
~~~~~~T1 new;~~
      A1 *addr = &E1;
      C2 val = (E2);
      C1 old = *addr;
      C1 new;
      do {
            new = old op val;
~~~~~~} while (!atomic_compare_exchange_strong(addr, &old, new));~~
      } while (!atomic_compare_exchange_weak(addr, &old, new));
```

with `new` being the result of the operation. The difference is that if the combination of the values of `old` and `val` is invalid for the operation, there will no signal raised or trap performed. In particular:

— If "`old` *op* `val`" has a signed type and produces an overflow, `new` is the corresponding modulo of the mathematical result of the operation.

— If the value of `val` is invalid for *op*, the value of `new` is unspecified.

— If `C2` is a pointer type and the value of `old` is null or is indeterminate, the value of `new` is unspecified.

— If `C2` is a pointer type and the value of "`old` *op* `val`" would be indeterminate, the value of `new` is unspecified.

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:

```
      #include <fenv.h>
      #pragma STDC FENV_ACCESS ON
      /* ... */
            fenv_t fenv;
~~~~~~~~~~~~T1 *addr = &E1;~~
~~~~~~~~~~~~T2 val = E2;~~
~~~~~~~~~~~~T1 old = *addr;~~
~~~~~~~~~~~~T1 new;~~
            A1 *addr = &E1;
            C2 val = E2;
            C1 old = *addr;
            C1 new;
            feholdexcept(&fenv);
            for (;;) {
                  new = old op val;
                  if (atomic_compare_exchange_strong(addr, &old, new))
                              break;
                  feclearexcept(FE_ALL_EXCEPT);
            }
            feupdateenv(&fenv);
```

If **`FLT_EVAL_METHOD`** is not 0, then ~~*T2*~~*C2* is expected to be a type with the range and precision to which `E2` is evaluated in order to satisfy the equivalence.

6.5.17 Comma operator

Syntax

1 *expression:*

assignment-expression
expression **,** *assignment-expression*

```
struct tnode s, *sp;
```

declares `s` to be an object of the given type and `sp` to be a pointer to an object of the given type. With these declarations, the expression `sp->left` refers to the left **`struct`** `tnode` pointer of the object to which `sp` points; the expression `s.right->count` designates the `count` member of the right **`struct`** `tnode` pointed to from `s`.

12 The following alternative formulation uses the **`typedef`** mechanism:

```
typedef struct tnode TNODE;
struct tnode {
      int count;
      TNODE *left, *right;
};
TNODE s, *sp;
```

13 **EXAMPLE 2** To illustrate the use of prior declaration of a tag to specify a pair of mutually referential structures, the declarations

```
struct s1 { struct s2 *s2p; /* ... */ }; // D1
struct s2 { struct s1 *s1p; /* ... */ }; // D2
```

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
```

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.[274)]

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);
```

[274)] 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.[276)]

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 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.

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

— An *M* refers to the type of the other argument for arithmetic operations. For atomic integer 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 It is unspecified whether any generic function declared in <stdatomic.h> 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.

7 **NOTE** 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~~ of these type generic functions have **volatile**-qualified parameters to allow their application to **volatile**-qualified objects.

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

7.17.2 Initialization

7.17.2.1 The ATOMIC_VAR_INIT macro

Synopsis

1
```
#include <stdatomic.h>
#define ATOMIC_VAR_INIT(C value)
```

Description

2 The **ATOMIC_VAR_INIT** macro 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.[277)]

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

4 **EXAMPLE**

```
atomic_int guide = ATOMIC_VAR_INIT(42);
```

7.17.2.2 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 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:[278)]

```
memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_acq_rel
```

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

```
    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** 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

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 volatile 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.[279)]

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~~,~~[280)] the atomic type name is declared as a type that has the same representation and alignment requirements as the ~~corresponding~~ direct type.[281)]

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)**

[279)] `obj` can be a null pointer.
[280)] See "future library directions" (7.31.10).
[281)] 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_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(volatile A *object, C desired);
void atomic_store_explicit(volatile 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 volatile A *object);
C atomic_load_explicit(const volatile 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(volatile A *object, C desired);
C atomic_exchange_explicit(volatile 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(volatile A *object, C *expected, C desired);
~~_Bool atomic_compare_exchange_strong_explicit(volatile A *object, C *expected,~~
~~        C desired, memory_order success, memory_order failure);~~
_Bool atomic_compare_exchange_strong_explicit(volatile A *object,
        C *expected, C desired, memory_order success, memory_order failure);
_Bool atomic_compare_exchange_weak(volatile A *object, C *expected, C desired);
~~_Bool atomic_compare_exchange_weak_explicit(volatile A *object, C *expected,~~
~~        C desired, memory_order success, memory_order failure);~~
_Bool atomic_compare_exchange_weak_explicit(volatile 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));
```

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.

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

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

The result of the comparison.

7.17.7.5 The `atomic_fetch` and modify generic functions

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`** . or pointer type, but not to **`atomic_bool`** or **`_Atomic(_Bool)`** , as long as the non-atomic version of the type can be the left operand of the corresponding `op=` compound assignment.[282)] The key, operator, and computation correspondence is:

key	op	computation
add	+	addition
sub	-	subtraction
or	|	bitwise inclusive or
xor	^	bitwise exclusive or
and	&	bitwise and

Synopsis

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

Description

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

Atomically, the value pointed to by `object` immediately before the effects. ~~The~~

NOTE 1 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~~

[282)] Thus these operations are not permitted for pointers to atomic **`_Bool`**, 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.

~~is the updated value of the object, whereas the~~ (6.5.16.2). Notable differences are: the value returned by the ~~is the previous value of the atomic object.~~ `atomic_fetch` and modify generic functions is the previous value of the atomic object; the memory order can be specified to be less strict than the operator; the set of operations does not include multiplicative and shift operators; the set of operands does not include Boolean and floating-point types; the possible range of values for `operand` can be different than for the operator.

6 **NOTE 2** The explicit forms of the `atomic_fetch` and modify generic functions are similar to the following definition:

```
C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order){
    C old = atomic_load_explicit(object, memory_order_relaxed);
    while (!atomic_compare_exchange_weak_explicit(
                object, &old,
                old op operand,
                order, memory_order_relaxed));
    return old;
}
```

As described in 6.5.16.2, for integer operands the operation "`old` *op* `operand`" will wrap in case of overflow, and for pointer operands will never raise a signal or perform a trap, and an unspecified value will be stored for an erroneous operation.

7 **NOTE 3** The non-explicit forms of the `atomic_fetch` and modify generic functions are the same only that `order` is omitted from the declaration and is replaced by **`memory_order_seq_cst`** in the compare-and-exchange operation. As a consequence only the last, successful, compare-and-exchange operation has **`memory_order_seq_cst`** memory order and is the only atomic operation that is visible in the total order of such **`memory_order_seq_cst`** operations.

8 **EXAMPLE** Provided that the implementation allows such large array sizes, the following use of the `+=` operator is valid. In contrast to that, in preparation of the arguments to the generic function call, a conversion of `large` to **`ptrdiff_t`** has to be performed. This may trap before the call or may give an implementation defined result that differs from the operator version.

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

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;
```

7.17.8.1 The `atomic_flag_test_and_set` functions

Synopsis

1

```
#include <stdatomic.h>
_Bool atomic_flag_test_and_set(volatile atomic_flag *object);
_Bool atomic_flag_test_and_set_explicit(volatile atomic_flag *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

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.

3 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.