Make false and true first-class language features

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 **N2458**

v 3

November 18, 2019

Make false and true first-class language features v3

proposal for C2x

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

In its London 2019 meeting, WG14 has found consensus to elevate **false** and **true** to proper keywords. By following the proposal of N2229 (vote in the Brno 2018 meeting of WG14), we should also change their type from **int** to **bool**.

Changes in v2: WG14 was not sympathetic to force these keywords also to be macros, so we remove the text corresponding to this idea. WG14 also was not in favor of the parts that proposed to introduce recommended practice and to add future language directions, so these are also removed.

Changes in v3: It was then observed in a discussion on the reflector, that the possible use of these predefined constants in the preprocessor needs some more precautions.

1. INTRODUCTION

The Boolean constants **false** and **true** are a bit ambivalent because in C17 they expand to integer constants `0` and `1` that have type **int** and not **_Bool**. This is unfortunate when they are used as arguments to type-generic macros, because there they could trigger an unexpected expansion, namely for **int** instead of **_Bool**. Since for C++, these constants are of type **bool**, we propose to do it the same.

The integration of these constants as proper language constructs, also allows to provide a better feedback to programmers, where such constants seem to be used erroneously. In particular, diagnostics may be provided when they are used in arithmetic or used contrary to the intent, *e.g* as null pointer constants.

2. IMPACT

The change should not have a big impact on user code. In most contexts where these constants are used (assignment, arithmetic, comparison, non-prototyped function call), **bool** values will be promoted to **int**, anyhow, and the property of being macro expansions (or not) should be transparent. So in these "regular" contexts the result after promotion would be exactly the same, namely **int** values 0 and 1, respectively. As arguments to function calls that provide a prototype, there is no change either, since the values 0 and 1 are valid for any arithmetic type and so the value and type received by the function are exactly the same.

The change can have marginal impact on existing code, when the constants are used in **sizeof**, **alignas** or **_Generic** expressions. All should be relatively rare. The latter, **_Generic**, is a sought effect of this change, because we think that choosing **bool** for these constants is a much more natural choice and will surprise less. In any case, these usages are compile-time detectable and we expect that quality implementations can provide diagnostics during the transition phase to C2x.

Another possible impact could be the use of these constants in preprocessing conditional expressions. Currently preprocessing arithmetic sees the existing macros from `<stdbool.h>` as signed values, and thus the result of expressions is merely consistent between the preprocessor and the rest of the language. When changing to keywords we should ensure that **false** and **true** may still be used in the preprocessor with the same semantics as before. This is done by enforcing the following:

— Other than other keywords **false** and **true** are not automatically rewritten to pp-token `0` in preprocessor arithmetic.
— We ensure that preprocessor arithmetic uses signed values for these constants, such that results of such arithmetic remain the same between C17 and C2x.

3. REFERENCE IMPLEMENTATION

To add minimal support for the proposed changes, an implementation that does not want to implement **false** and **true** as full-featured keywords would have to add definitions that are equivalent to the following lines to their startup code:

```
#define false          ((bool)+0)
#define true           ((bool)+1)
```

Notice that these do not use the literals `0U` or `1U` because with that arithmetic with these constants in the preprocessor would be performed as unsigned integers. This would have the consequence that something like `-true` would result to **UINTMAX_MAX** in the preprocessor and `-1` otherwise.

4. CHANGES

Predefined constants need a little bit more effort for the integration, than the other keywords in N2457, because up to now C did not have named constants on the level of the language. We propose to integrate these constants by means of a new syntax term `predefined constant`. Besides minor word replacements the proposed changes consist of the following:

— Add the constants to the list of keywords in 6.4.1.
— Add the "predefined constant" syntax term and a new clause 6.4.4.5 that describes it.
— Add a specific clause for the two constants, 6.4.4.5.1.
— Exempt these constants from being replaced in preprocessor arithmetic, 6.10.1
— Add them to the optional predefined macros, 6.10.8.4.
— Adapt the text for `<stdbool.h>` (7.18) and make this header obsolescent.

Appendix: pages with diffmarks of the proposed changes against modifications proposed in N2457.

The following page numbers are from the particular snapshot and may vary once the changes are integrated.

6.2.5 Types

1 The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. (An identifier declared to be an object is the simplest such expression; the type is specified in the declaration of the identifier.) Types are partitioned into *object types* (types that describe objects) and *function types* (types that describe functions). At various points within a translation unit an object type may be *incomplete* (lacking sufficient information to determine the size of objects of that type) or *complete* (having sufficient information).[39)]

2 An object declared as type **bool** is large enough to store the values ~~0 and 1~~**false** and **true** .

3 An object declared as type **char** is large enough to store any member of the basic execution character set. If a member of the basic execution character set is stored in a **char** object, its value is guaranteed to be nonnegative. If any other character is stored in a **char** object, the resulting value is implementation-defined but shall be within the range of values that can be represented in that type.

4 There are five *standard signed integer types*, designated as **signed char**, **short int**, **int**, **long int**, and **long long int**. (These and other types may be designated in several additional ways, as described in 6.7.2.) There may also be implementation-defined *extended signed integer types*.[40)] The standard and extended signed integer types are collectively called *signed integer types*.[41)]

5 An object declared as type **signed char** occupies the same amount of storage as a "plain" **char** object. A "plain" **int** object has the natural size suggested by the architecture of the execution environment (large enough to contain any value in the range **INT_MIN** to **INT_MAX** as defined in the header `<limits.h>`).

6 For each of the signed integer types, there is a *corresponding* (but different) *unsigned integer type* (designated with the keyword **unsigned**) that uses the same amount of storage (including sign information) and has the same alignment requirements. The type **bool** and the unsigned integer types that correspond to the standard signed integer types are the *standard unsigned integer types*. The unsigned integer types that correspond to the extended signed integer types are the *extended unsigned integer types*. The standard and extended unsigned integer types are collectively called *unsigned integer types*.[42)]

7 The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*; the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*.

8 For any two integer types with the same signedness and different integer conversion rank (see 6.3.1.1), the range of values of the type with smaller integer conversion rank is a subrange of the values of the other type.

9 The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same.[43)] A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

10 There are three *standard floating types*, designated as **float**, **double**, and **long double**.[44)] The set of values of the type **float** is a subset of the set of values of the type **double**; the set of values of the type **double** is a subset of the set of values of the type **long double**.

[39)] A type can be incomplete or complete throughout an entire translation unit, or it can change states at different points within a translation unit.

[40)] Implementation-defined keywords have the form of an identifier reserved for any use as described in 7.1.3.

[41)] Therefore, any statement in this document about signed integer types also applies to the extended signed integer types.

[42)] Therefore, any statement in this document about unsigned integer types also applies to the extended unsigned integer types.

[43)] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

[44)] See "future language directions" (6.11.1).

— The rank of **long long int** shall be greater than the rank of **long int**, which shall be greater than the rank of **int**, which shall be greater than the rank of **short int**, which shall be greater than the rank of **signed char**.

— The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.

— The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.

— The rank of **char** shall equal the rank of **signed char** and **unsigned char**.

— The rank of **bool** shall be less than the rank of all other standard integer types.

— The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2).

— The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank.

— For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3, then T1 has greater rank than T3.

2 The following may be used in an expression wherever an **int** or **unsigned int** may be used:

— An object or expression with an integer type (other than **int** or **unsigned int**) whose integer conversion rank is less than or equal to the rank of **int** and **unsigned int**.

— A bit-field of type **bool**, **int**, **signed int**, or **unsigned int**.

If an **int** can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an **int**; otherwise, it is converted to an **unsigned int**. These are called the *integer promotions*.[62] All other types are unchanged by the integer promotions.

3 The integer promotions preserve value including sign. As discussed earlier, whether a "plain" **char** can hold negative values is implementation-defined.

Forward references: enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1).

6.3.1.2 Boolean type

1 When any scalar value is converted to **bool**, the result is ~~0~~ **false** if the value compares equal to 0; otherwise, the result is ~~1~~**true** .[63]

6.3.1.3 Signed and unsigned integers

1 When a value with integer type is converted to another integer type other than **bool**, if the value can be represented by the new type, it is unchanged.

2 Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.[64]

3 Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

[62] The integer promotions are applied only: as part of the usual arithmetic conversions, to certain argument expressions, to the operands of the unary +, - , and ~ operators, and to both operands of the shift operators, as specified by their respective subclauses.

[63] NaNs do not compare equal to 0 and thus convert to **true**.

[64] The rules describe arithmetic on the mathematical value, not the value of a given type of expression.

6.4.1 Keywords

Syntax

1 *keyword:* one of

alignas	**enum**	**short**	**volatile**
alignof	**extern**	**signed**	**while**
auto	**false**	**sizeof**	**_Atomic**
bool	**float**	**static**	**_Complex**
break	**for**	**static_assert**	**_Decimal128**
case	**goto**	**struct**	**_Decimal32**
char	**if**	**switch**	**_Decimal64**
const	**inline**	**thread_local**	**_Generic**
continue	**int**	**true**	**_Imaginary**
default	**long**	**typedef**	**_Noreturn**
do	**register**	**union**	
double	**restrict**	**unsigned**	
else	**return**	**void**	

Constraints

2 The keywords

alignas	**bool**	**static_assert**	**true**
alignof	**false**	**thread_local**	

may optionally be predefined macro names (6.10.8.4). None of these shall be the subject of a **#define** or a **#undef** preprocessing directive.

Semantics

3 The above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords except in an attribute token, and shall not be used otherwise. The keyword **_Imaginary** is reserved for specifying imaginary types.[74)]

4 The following table provides alternate spellings for certain keywords. These can be used wherever the keyword can.[75)]

keyword	alternative spelling
alignas	**_Alignas**
alignof	**_Alignof**
bool	**_Bool**
static_assert	**_Static_assert**
thread_local	**_Thread_local**

~~Their spelling inside expressions~~ The spelling of keywords that are also predefined macros and that are subject to the **#** and **##** preprocessing operators is unspecified.[76)]

6.4.2 Identifiers

6.4.2.1 General

Syntax

1 *identifier:*

identifier-nondigit
identifier identifier-nondigit
identifier digit

[74)] One possible specification for imaginary types appears in Annex G.
[75)] These alternative keywords are obsolescent features and should not be used for new code.
[76)] The intent of these specifications is to allow but not to force the implementation of the correspondig feature by means of a predefined macro.

6.4.4 Constants

Syntax

1 *constant:*
integer-constant
floating-constant
enumeration-constant
character-constant
predefined-constant

Constraints

2 Each constant shall have a type and the value of a constant shall be in the range of representable values for its type.

Semantics

3 Each constant has a type, determined by its form and value, as detailed later.

6.4.4.1 Integer constants

Syntax

1 *integer-constant:*
decimal-constant *integer-suffix*$_{\text{opt}}$
octal-constant *integer-suffix*$_{\text{opt}}$
hexadecimal-constant *integer-suffix*$_{\text{opt}}$

decimal-constant:
nonzero-digit
decimal-constant *digit*

octal-constant:
0
octal-constant *octal-digit*

hexadecimal-constant:
hexadecimal-prefix *hexadecimal-digit*
hexadecimal-constant *hexadecimal-digit*

hexadecimal-prefix: one of
0x 0X

nonzero-digit: one of
1 2 3 4 5 6 7 8 9

octal-digit: one of
0 1 2 3 4 5 6 7

hexadecimal-digit: one of
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

integer-suffix:
unsigned-suffix *long-suffix*$_{\text{opt}}$
unsigned-suffix *long-long-suffix*
long-suffix *unsigned-suffix*$_{\text{opt}}$
long-long-suffix *unsigned-suffix*$_{\text{opt}}$

15 **EXAMPLE 2** Consider implementations that use two's complement representation for integers and eight bits for objects that have type **char**. In an implementation in which type **char** has the same range of values as **signed char**, the integer character constant `'\xFF'` has the value −1; if type **char** has the same range of values as **unsigned char**, the character constant `'\xFF'` has the value +255.

16 **EXAMPLE 3** Even if eight bits are used for objects that have type **char**, the construction `'\x123'` specifies an integer character constant containing only one character, since a hexadecimal escape sequence is terminated only by a non-hexadecimal character. To specify an integer character constant containing the two characters whose values are `'\x12'` and `'3'`, the construction `'\0223'` can be used, since an octal escape sequence is terminated after three octal digits. (The value of this two-character integer character constant is implementation-defined.)

17 **EXAMPLE 4** Even if 12 or more bits are used for objects that have type **wchar_t**, the construction `L'\1234'` specifies the implementation-defined value that results from the combination of the values `0123` and `'4'`.

Forward references: common definitions <stddef.h> (7.19), the **mbtowc** function (7.22.7.2), Unicode utilities <uchar.h> (7.28).

6.4.4.5 Predefined constants

Syntax

1 *predefined-constant:*
false
true

Description

Some keywords represent constants of a specific value and type.

6.4.4.5.1 The **false** and **true** constants

Description

1 The keywords **false** and **true** represent constants of type **bool** that are suitable for use as are integer literals. Their values are 0 for **false** and 1 for **true**.[87] When used in preprocessor conditional expressions, the keywords **false** and **true** behave as if replaced with the pp-numbers `0` and `1`, respectively.[88]

6.4.5 String literals

Syntax

1 *string-literal:*
encoding-prefix$_{\text{opt}}$ **"** *s-char-sequence*$_{\text{opt}}$ **"**
s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except
the double-quote `"`, backslash `\`, or new-line character
escape-sequence

Constraints

2 A sequence of adjacent string literal tokens shall not include both a wide string literal and a UTF–8 string literal.

[87] When used in arithmetic expressions after translation phase 4 the values of the keywords are promoted to type **int**.
[88] Therefore, arithmetic with **false** and **true** in translation phase 4 presents results that are generally consistent with later translation phases.

token (6.4).

Semantics

3 Preprocessing directives of the forms

 # if *constant-expression new-line* $group_{opt}$
 # elif *constant-expression new-line* $group_{opt}$

check whether the controlling constant expression evaluates to nonzero.

4 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the **defined** unary operator), just as in normal text. If the token **defined** is generated as a result of this replacement process or use of the **defined** unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the **defined** unary operator have been performed, all remaining identifiers other than **false** and **true** (including those lexically identical to keywords) are replaced with the pp-number `0`, and then each preprocessing token is converted into a token. The resulting tokens compose the controlling constant expression which is evaluated according to the rules of 6.6. For the purposes of this token conversion and evaluation, all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, the types **intmax_t** and **uintmax_t** defined in the header `<stdint.h>`.[181] This includes interpreting character constants, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a **#if** or **#elif** directive) is implementation-defined.[182] Also, whether a single-character character constant may have a negative value is implementation-defined.

5 Preprocessing directives of the forms

 # ifdef *identifier new-line* $group_{opt}$
 # ifndef *identifier new-line* $group_{opt}$

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to **#if defined** *identifier* and **#if !defined** *identifier* respectively.

6 Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed; any following groups are skipped and their controlling directives are processed as if they were in a group that is skipped. If none of the conditions evaluates to true, and there is a **#else** directive, the group controlled by the **#else** is processed; lacking a **#else** directive, all the groups until the **#endif** are skipped.[183]

Forward references: macro replacement (6.10.3), source file inclusion (6.10.2), largest integer types (7.20.1.5).

[181] Thus, on an implementation where **INT_MAX** is `0x7FFF` and **UINT_MAX** is `0xFFFF`, the constant `0x8000` is signed and positive within a **#if** expression even though it would be unsigned in translation phase 7.

[182] Thus, the constant expression in the following **#if** directive and **if** statement is not guaranteed to evaluate to the same value in these two contexts.

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

[183] As indicated by the syntax, no preprocessing tokens are allowed to follow a **#else** or **#endif** directive before the terminating new-line character. However, comments can appear anywhere in a source file, including within a preprocessing directive.

2 An implementation that defines **__STDC_NO_COMPLEX__** shall not define **__STDC_IEC_60559_COMPLEX__** or **__STDC_IEC_559_COMPLEX__**.

6.10.8.4 Optional macros

1 The keywords

alignas	**bool**	**static_assert**	**true**
alignof	**false**	**thread_local**	

optionally are also predefined macro names that expand to unspecified tokens.

6.10.9 Pragma operator

Semantics

1 A unary operator expression of the form:

_Pragma (*string-literal* **)**

is processed as follows: The string literal is *destringized* by deleting any encoding prefix, deleting the leading and trailing double-quotes, replacing each escape sequence \" by a double-quote, and replacing each escape sequence \\ by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

2 **EXAMPLE** A directive of the form:

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ("listing on \"..\\listing.dir\"")
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x)  _Pragma(#x)

LISTING (..\listing.dir)
```

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 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.

Recommended practice

2 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. The implementation should not depend on any per-process state. This restriction enables communication via memory mapped into a process more than once and memory shared between two processes.

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)~~ **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.[280)]

7.17.6 Atomic integer types

1 For each line in the following table,[281)] the atomic type name is declared as a type that has the same representation and alignment requirements as the corresponding direct type.[282)]

Atomic type name	Direct type
atomic_bool	**_Atomic bool**
atomic_char	**_Atomic char**
atomic_schar	**_Atomic signed char**
atomic_uchar	**_Atomic unsigned char**
atomic_short	**_Atomic short**
atomic_ushort	**_Atomic unsigned short**
atomic_int	**_Atomic int**
atomic_uint	**_Atomic unsigned int**
atomic_long	**_Atomic long**
atomic_ulong	**_Atomic unsigned long**
atomic_llong	**_Atomic long long**
atomic_ullong	**_Atomic unsigned long long**
atomic_char16_t	**_Atomic char16_t**
atomic_char32_t	**_Atomic char32_t**
atomic_wchar_t	**_Atomic wchar_t**
atomic_int_least8_t	**_Atomic int_least8_t**
atomic_uint_least8_t	**_Atomic uint_least8_t**
atomic_int_least16_t	**_Atomic int_least16_t**

[280)] obj can be a null pointer.
[281)] See "future library directions" (7.31.11).
[282)] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

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_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);
```

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 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~~ `false` 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.

```
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 `order`. These operations are atomic read-modify-write operations (5.1.2.4).

Returns

3 The **`atomic_flag_test_and_set`** 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` functions

Synopsis

1

```
#include <stdatomic.h>
void atomic_flag_clear(volatile atomic_flag *object);
void atomic_flag_clear_explicit(volatile atomic_flag *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`** functions return no value.

7.18 Boolean type and values <stdbool.h>

1 The obsolescent header <stdbool.h> defines ~~three macros that are~~ the following macro which is suitable for use in **#if** ~~preprocessing directives. They are which expands to the integer constant 1, which expands to the integer constant 0, and~~ conditional preprocessing directives:

```
__bool_true_false_are_defined
```

~~which~~ It expands to the ~~integer constant 1~~constant **true** .

~~Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros bool, **true**, and **false**.~~

cracosh	cratanh	crexp10	crlog1p	crrootn
cracospi	cratanpi	crexp2m1	crlog2p1	crrsqrt
cracos	cratan	crexp2	crlog2	crsinh
crasinh	crcompoundn	crexpm1	crlogp1	crsinpi
crasinpi	crcosh	crexp	crlog	crsin
crasin	crcospi	crhypot	crpown	crtanh
cratan2pi	crcos	crlog10p1	crpowr	crtanpi
cratan2	crexp10m1	crlog10	crpow	crtan

and the same names suffixed with **f**, **l**, **d32**, **d64**, or **d128** may be added to the <math.h> header. The **cr** prefix is intended to indicate a correctly rounded version of the function.

7.31.9 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.

7.31.10 Alignment <stdalign.h>

1 The header <stdalign.h> together with its defined macros **__alignas_is_defined** and **__alignas_is_defined** is an obsolescent feature.

7.31.11 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.

7.31.12 Boolean type and values <stdbool.h>

1 The ~~ability to undefine and perhaps then redefine the macros **true**, and **false**~~ header <stdbool.h> together with its defined macro **__bool_true_false_are_defined** is an obsolescent feature.

7.31.13 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.

7.31.14 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.

7.31.15 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.

7.31.16 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.

Annex A
(informative)
Language syntax summary

1 **NOTE** The notation is described in 6.1.

A.1 Lexical grammar

A.1.1 Lexical elements

(6.4) *token:*
keyword
identifier
constant
string-literal
punctuator

(6.4) *preprocessing-token:*
header-name
identifier
pp-number
character-constant
string-literal
punctuator
each non-white-space character that cannot be one of the above

A.1.2 Keywords

(6.4.1) *keyword:* one of

alignas	**enum**	**short**	**volatile**
alignof	**extern**	**signed**	**while**
auto	**false**	**sizeof**	**_Atomic**
bool	**float**	**static**	**_Complex**
break	**for**	**static_assert**	**_Decimal128**
case	**goto**	**struct**	**_Decimal32**
char	**if**	**switch**	**_Decimal64**
const	**inline**	**thread_local**	**_Generic**
continue	**int**	**true**	**_Imaginary**
default	**long**	**typedef**	**_Noreturn**
do	**register**	**union**	
double	**restrict**	**unsigned**	
else	**return**	**void**	

A.1.3 Identifiers

(6.4.2.1) *identifier:*
identifier-nondigit
identifier identifier-nondigit
identifier digit

(6.4.2.1) *identifier-nondigit:*
nondigit
universal-character-name
other implementation-defined characters

(6.4.2.1) *nondigit:* one of

```
_ a b c d e f g h i j k l m
  n o p q r s t u v w x y z
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z
```

(6.4.2.1) *digit:* one of

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

A.1.4 Universal character names

(6.4.3) *universal-character-name:*

\u *hex-quad*
\U *hex-quad hex-quad*

(6.4.3) *hex-quad:*

hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

A.1.5 Constants

(6.4.4) *constant:*

integer-constant
floating-constant
enumeration-constant
character-constant
predefined-constant

(6.4.4.1) *integer-constant:*

decimal-constant integer-suffix$_{\text{opt}}$
octal-constant integer-suffix$_{\text{opt}}$
hexadecimal-constant integer-suffix$_{\text{opt}}$

(6.4.4.1) *decimal-constant:*

nonzero-digit
decimal-constant digit

(6.4.4.1) *octal-constant:*

0
octal-constant octal-digit

(6.4.4.1) *hexadecimal-constant:*

hexadecimal-prefix hexadecimal-digit
hexadecimal-constant hexadecimal-digit

(6.4.4.1) *hexadecimal-prefix:* one of

```
0x 0X
```

(6.4.4.1) *nonzero-digit:* one of

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

(6.4.4.1) *octal-digit:* one of

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

(6.4.4.1) *hexadecimal-digit:* one of

```
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F
```

(6.4.4.3) *enumeration-constant:*
identifier
(6.4.4.4) *character-constant:*
encoding-prefix$_{\text{opt}}$ **'** *c-char-sequence* **'**
(6.4.4.4) *encoding-prefix:*
u8
u
U
L

(6.4.4.4) *c-char-sequence:*
c-char
c-char-sequence c-char
(6.4.4.4) *c-char:*
any member of the source character set except
the single-quote **'**, backslash ****, or new-line character
escape-sequence

(6.4.4.4) *escape-sequence:*
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence
universal-character-name

(6.4.4.4) *simple-escape-sequence:* one of
**\' \" \? **
\a \b \f \n \r \t \v

(6.4.4.4) *octal-escape-sequence:*
**** *octal-digit*
**** *octal-digit octal-digit*
**** *octal-digit octal-digit octal-digit*

(6.4.4.4) *hexadecimal-escape-sequence:*
\x *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

A.1.5.1 Predefined constants

(6.4.4.5) *predefined-constant:*
false
true

A.1.6 String literals

(6.4.5) *string-literal:*
encoding-prefix$_{\text{opt}}$ **"** *s-char-sequence*$_{\text{opt}}$ **"**
(6.4.5) *s-char-sequence:*
s-char
s-char-sequence s-char

(6.4.5) *s-char:*
any member of the source character set except
the double-quote **"**, backslash ****, or new-line character
escape-sequence

Annex H
(informative)
Language independent arithmetic

H.1 Introduction

1 This annex documents the extent to which the C language supports the ISO/IEC 10967–1 standard for language-independent arithmetic (LIA–1). LIA–1 is more general than IEC 60559 (Annex F) in that it covers integer and diverse floating-point arithmetics.

H.2 Types

1 The relevant C arithmetic types meet the requirements of LIA–1 types if an implementation adds notification of exceptional arithmetic operations and meets the 1 unit in the last place (ULP) accuracy requirement (LIA–1 subclause 5.2.8).

H.2.1 Boolean type

1 The LIA–1 data type Boolean is implemented by the C data type **`bool`** with values of **`true`** and **`false`** ~~, all from .~~.

H.2.2 Integer types

1 The signed C integer types **`int`**, **`long int`**, **`long long int`**, and the corresponding unsigned types are compatible with LIA–1. If an implementation adds support for the LIA–1 exceptional values "integer_overflow" and "undefined", then those types are LIA–1 conformant types. C's unsigned integer types are "modulo" in the LIA–1 sense in that overflows or out-of-bounds results silently wrap. An implementation that defines signed integer types as also being modulo need not detect integer overflow, in which case, only integer divide-by-zero need be detected.

2 The parameters for the integer data types can be accessed by the following:

maxint **`INT_MAX`**, **`LONG_MAX`**, **`LLONG_MAX`**, **`UINT_MAX`**, **`ULONG_MAX`**, **`ULLONG_MAX`**

minint **`INT_MIN`**, **`LONG_MIN`**, **`LLONG_MIN`**

3 The parameter "bounded" is always true, and is not provided. The parameter "minint" is always 0 for the unsigned types, and is not provided for those types.

H.2.2.1 Integer operations

1 The integer operations on integer types are the following:

addI `x + y`

subI `x - y`

mulI `x * y`

divI, divtI `x / y`

remI, remtI `x % y`

negI `-x`

absI **`abs`**`(x),`**`labs`**`(x),`**`llabs`**`(x)`

eqI `x == y`

neqI `x != y`

lssI `x < y`

leqI `x <= y`

Annex M
(informative)
Change History

M.1 Fifth Edition

1 Major changes in this fifth edition (__STDC_VERSION__ *yyyymm*L) include:

— add a one-argument version of **static_assert**, make it a keyword and deprecate the underscore-capital form

— support for function definitions with identifier lists has been removed

— harmonization with ISO/IEC 9945 (POSIX):

- extended month name formats for **strftime**
- integration of functions: **asctime_r**, **ctime_r**, **gmtime_r**, **localtime_r**, **memccpy**, **strdup**, **strndup**

— harmonization with floating point standard IEC 60559:

- integration of binary floating-point technical specification TS 18661-1
- integration of decimal floating-point technical specification TS 18661-2
- integration of decimal floating-point technical specification TS 18661-4a

— the macro **DECIMAL_DIG** is declared obsolescent

— added version test macros to certain library headers

— added the attributes feature

— added **deprecated**, **fallthrough**, **maybe_unused**, and **nodiscard** attributes

— added the u8 character prefix

— change **bool**, **alignas**, **alignof** and **thread_local** to be keywords and deprecate the underscore-capital forms

— change **false** and **true** to keywords and make them type **bool**

M.2 Fourth Edition

1 There were no major changes in the fourth edition (__STDC_VERSION__ 201710L), only technical corrections and clarifications.

M.3 Third Edition

1 Major changes in the third edition (__STDC_VERSION__ 201112L) included:

— conditional (optional) features (including some that were previously mandatory)

— support for multiple threads of execution including an improved memory sequencing model, atomic objects, and thread-local storage (<stdatomic.h> and <threads.h>)

— additional floating-point characteristic macros (<float.h>)

— querying and specifying alignment of objects (<stdalign.h>, <stdlib.h>)

— Unicode characters and strings (<uchar.h>) (originally specified in ISO/IEC TR 19769:2004)

— type-generic expressions

— static assertions