INTERNATIONAL STANDARD ©ISO/IEC **ISO/IEC 9899:201x**

Programming languages — C

ABSTRACT

(Cover sheet to be provided by ISO Secretariat.)

This International Standard specifies the form and establishes the interpretation of programs expressed in the programming language C. Its purpose is to promote portability, reliability, maintainability, and efficient execution of C language programs on a variety of computing systems.

Clauses are included that detail the C language itself and the contents of the C language execution library. Annexes summarize aspects of both of them, and enumerate factors that influence the portability of C programs.

Although this International Standard is intended to guide knowledgeable C language programmers as well as implementors of C language translation systems, the document itself is not designed to serve as a tutorial.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Changes from the previous draft (N1539) are indicated by ‘‘diff marks’’ in the right margin: deleted text is marked with ‘‘*’’, new or changed text with ‘‘|’’.

Contents

Foreword

1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are member of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

2 International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2. This International Standard was drafted in accordance with the fifth edition (2004).

3 In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

4 Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

5 This International Standard was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*. The Working Group responsible for this standard (WG 14) maintains a site on the World Wide Web at **`http://www.open-std.org/JTC1/SC22/WG14/`** containing additional information relevant to this standard such as a Rationale for many of the decisions made during its preparation and a log of Defect Reports and Responses.

6 This third edition cancels and replaces the second edition, ISO/IEC 9899:1999, as corrected by ISO/IEC 9899:1999/Cor 1:2001, ISO/IEC 9899:1999/Cor 2:2004, and ISO/IEC 9899:1999/Cor 3:2007. Major changes from the previous edition include:

— 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

— anonymous structures and unions

— no-return functions

— macros to create complex numbers (**<complex.h>**)

— support for opening files for exclusive access

— removed the **gets** function (**<stdio.h>**)

— added the **aligned_alloc**, **at_quick_exit**, and **quick_exit** functions (**<stdlib.h>**)

— (conditional) support for bounds-checking interfaces (originally specified in ISO/IEC TR 24731−1:2007)

— (conditional) support for analyzability

7 Major changes in the second edition included:

— restricted character set support via digraphs and **<iso646.h>** (originally specified in AMD1)

— wide character library support in **<wchar.h>** and **<wctype.h>** (originally specified in AMD1)

— more precise aliasing rules via effective type

— restricted pointers

— variable length arrays

— flexible array members

— **static** and type qualifiers in parameter array declarators

— complex (and imaginary) support in **<complex.h>**

— type-generic math macros in **<tgmath.h>**

— the **long long int** type and library functions

— increased minimum translation limits

— additional floating-point characteristics in **<float.h>**

— remove implicit **int**

— reliable integer division

— universal character names (**\u** and **\U**)

— extended identifiers

— hexadecimal floating-point constants and **%a** and **%A** **printf**/**scanf** conversion specifiers

— compound literals

— designated initializers

— `//` comments

— extended integer types and library functions in `<inttypes.h>` and `<stdint.h>`

— remove implicit function declaration

— preprocessor arithmetic done in `intmax_t`/`uintmax_t`

— mixed declarations and code

— new block scopes for selection and iteration statements

— integer constant type rules

— integer promotion rules

— macros with a variable number of arguments

— the `vscanf` family of functions in `<stdio.h>` and `<wchar.h>`

— additional math library functions in `<math.h>`

— treatment of error conditions by math library functions (`math_errhandling`)

— floating-point environment access in `<fenv.h>`

— IEC 60559 (also known as IEC 559 or IEEE arithmetic) support

— trailing comma allowed in `enum` declaration

— `%lf` conversion specifier allowed in `printf`

— inline functions

— the `snprintf` family of functions in `<stdio.h>`

— boolean type in `<stdbool.h>`

— idempotent type qualifiers

— empty macro arguments

— new structure type compatibility rules (tag compatibility)

— additional predefined macro names

— `_Pragma` preprocessing operator

— standard pragmas

— `__func__` predefined identifier

— `va_copy` macro

— additional `strftime` conversion specifiers

— LIA compatibility annex

— deprecate `ungetc` at the beginning of a binary file

— remove deprecation of aliased array parameters

— conversion of array to pointer not limited to lvalues

— relaxed constraints on aggregate and union initialization

— relaxed restrictions on portable header names

— `return` without expression not permitted in function that returns a value (and vice versa)

8 Annexes D, F, G, K, and L form a normative part of this standard; annexes A, B, C, E, H, I, J, the bibliography, and the index are for information only. In accordance with Part 2 of the ISO/IEC Directives, this foreword, the introduction, notes, footnotes, and examples are also for information only.

Introduction

1 With the introduction of new devices and extended character sets, new features may be added to this International Standard. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, may conflict with future additions.

2 Certain features are *obsolescent*, which means that they may be considered for withdrawal in future revisions of this International Standard. They are retained because of their widespread use, but their use in new implementations (for implementation features) or new programs (for language [6.11] or library features [7.31]) is discouraged.

3 This International Standard is divided into four major subdivisions:

— preliminary elements (clauses 1−4);

— the characteristics of environments that translate and execute C programs (clause 5);

— the language syntax, constraints, and semantics (clause 6);

— the library facilities (clause 7).

4 Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in that subclause or elsewhere in this International Standard. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementors. Annexes provide additional information and summarize the information contained in this International Standard. A bibliography lists documents that were referred to during the preparation of the standard.

5 The language clause (clause 6) is derived from ‘‘The C Reference Manual’’.

6 The library clause (clause 7) is based on the *1984 /usr/group Standard*.

Programming languages — C

1. Scope

1 This International Standard specifies the form and establishes the interpretation of programs written in the C programming language.[1)] It specifies

— the representation of C programs;

— the syntax and constraints of the C language;

— the semantic rules for interpreting C programs;

— the representation of input data to be processed by C programs;

— the representation of output data produced by C programs;

— the restrictions and limits imposed by a conforming implementation of C.

2 This International Standard does not specify

— the mechanism by which C programs are transformed for use by a data-processing system;

— the mechanism by which C programs are invoked for use by a data-processing system;

— the mechanism by which input data are transformed for use by a C program;

— the mechanism by which output data are transformed after being produced by a C program;

— the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor;

— all minimal requirements of a data-processing system that is capable of supporting a conforming implementation.

1) This International Standard is designed to promote the portability of C programs among a variety of data-processing systems. It is intended for use by implementors and programmers.

2. Normative references

1 The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2 ISO 31−11:1992, *Quantities and units — Part 11: Mathematical signs and symbols for use in the physical sciences and technology*.

3 ISO/IEC 646, *Information technology — ISO 7-bit coded character set for information interchange*.

4 ISO/IEC 2382−1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*.

5 ISO 4217, *Codes for the representation of currencies and funds*.

6 ISO 8601, *Data elements and interchange formats — Information interchange — Representation of dates and times*.

7 ISO/IEC 10646 (all parts), *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*.

8 IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* (previously designated IEC 559:1989).

3. Terms, definitions, and symbols

1 For the purposes of this International Standard, the following definitions apply. Other terms are defined where they appear in *italic* type or on the left side of a syntax rule. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this International Standard are to be interpreted according to ISO/IEC 2382−1. Mathematical symbols not defined in this International Standard are to be interpreted according to ISO 31−11.

3.1

1 **access**

⟨execution-time action⟩ to read or modify the value of an object

2 NOTE 1 Where only one of these two actions is meant, ‘‘read’’ or ‘‘modify’’ is used.

3 NOTE 2 ‘‘Modify’’ includes the case where the new value being stored is the same as the previous value.

4 NOTE 3 Expressions that are not evaluated do not access objects.

3.2

1 **alignment**

requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address

3.3

1 **argument**

actual argument

actual parameter (deprecated)

expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation

3.4

1 **behavior**

external appearance or action

3.4.1

1 **implementation-defined behavior**

unspecified behavior where each implementation documents how the choice is made

2 EXAMPLE An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.

3.4.2

1 **locale-specific behavior**

behavior that depends on local conventions of nationality, culture, and language that each implementation documents

2 EXAMPLE An example of locale-specific behavior is whether the **`islower`** function returns true for characters other than the 26 lowercase Latin letters.

3.4.3

1 **undefined behavior**

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

2 NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

3 EXAMPLE An example of undefined behavior is the behavior on integer overflow.

3.4.4

1 **unspecified behavior**

use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

2 EXAMPLE An example of unspecified behavior is the order in which the arguments to a function are evaluated.

3.5

1 **bit**

unit of data storage in the execution environment large enough to hold an object that may have one of two values

2 NOTE It need not be possible to express the address of each individual bit of an object.

3.6

1 **byte**

addressable unit of data storage large enough to hold any member of the basic character set of the execution environment

2 NOTE 1 It is possible to express the address of each individual byte of an object uniquely.

3 NOTE 2 A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*.

3.7

1 **character**

⟨abstract⟩ member of a set of elements used for the organization, control, or representation of data

3.7.1

1 **character**

single-byte character

⟨C⟩ bit representation that fits in a byte

3.7.2

1 **multibyte character**
sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment

2 NOTE The extended character set is a superset of the basic character set.

3.7.3

1 **wide character**
value representable by an object of type **`wchar_t`**, capable of representing any character in the current locale

3.8

1 **constraint**
restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted

3.9

1 **correctly rounded result**
representation in the result format that is nearest in value, subject to the current rounding mode, to what the result would be given unlimited range and precision

3.10

1 **diagnostic message**
message belonging to an implementation-defined subset of the implementation's message output

3.11

1 **forward reference**
reference to a later subclause of this International Standard that contains additional information relevant to this subclause

3.12

1 **implementation**
particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment

3.13

1 **implementation limit**
restriction imposed upon programs by the implementation

3.14

1 **memory location**
either an object of scalar type, or a maximal sequence of adjacent bit-fields all having nonzero width

2 NOTE 1 Two threads of execution can update and access separate memory locations without interfering with each other.

3 NOTE 2 A bit-field and an adjacent non-bit-field member are in separate memory locations. The same applies to two bit-fields, if one is declared inside a nested structure declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field member declaration. It is not safe to concurrently update two non-atomic bit-fields in the same structure if all members declared between them are also (non-zero-length) bit-fields, no matter what the sizes of those intervening bit-fields happen to be.

4 EXAMPLE A structure declared as

```
struct {
        char a;
        int b:5, c:11, :0, d:8;
        struct { int ee:8; } e;
}
```

contains four separate memory locations: The member **`a`**, and bit-fields **`d`** and **`e.ee`** are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields **`b`** and **`c`** together constitute the fourth memory location. The bit-fields **`b`** and **`c`** cannot be concurrently modified, but **`b`** and **`a`**, for example, can be.

3.15

1 **object**

region of data storage in the execution environment, the contents of which can represent values

2 NOTE When referenced, an object may be interpreted as having a particular type; see 6.3.2.1.

3.16

1 **parameter**

formal parameter

formal argument (deprecated)

object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition

3.17

1 **recommended practice**

specification that is strongly recommended as being in keeping with the intent of the standard, but that may be impractical for some implementations

3.18

1 **runtime-constraint**

requirement on a program when calling a library function

2 NOTE 1 Despite the similar terms, a runtime-constraint is not a kind of constraint as defined by 3.8, and need not be diagnosed at translation time.

3 NOTE 2 Implementations that support the extensions in annex K are required to verify that the runtime-constraints for a library function are not violated by the program; see K.3.1.4.

3.19

1 **value**

precise meaning of the contents of an object when interpreted as having a specific type

3.19.1

1 **implementation-defined value**

unspecified value where each implementation documents how the choice is made

3.19.2

1 **indeterminate value**

either an unspecified value or a trap representation

3.19.3

1 **unspecified value**

valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance

2 NOTE An unspecified value cannot be a trap representation.

3.19.4

1 **trap representation**

an object representation that need not represent a value of the object type

3.19.5

1 **perform a trap**

interrupt execution of the program such that no further operations are performed

2 NOTE In this International Standard, when the word ‘‘trap’’ is not immediately followed by ‘‘representation’’, this is the intended usage.[2)]

3.20

1 $\lceil x \rceil$

ceiling of x: the least integer greater than or equal to x

2 EXAMPLE $\lceil 2.4 \rceil$ is 3, $\lceil -2.4 \rceil$ is -2.

3.21

1 $\lfloor x \rfloor$

floor of x: the greatest integer less than or equal to x

2 EXAMPLE $\lfloor 2.4 \rfloor$ is 2, $\lfloor -2.4 \rfloor$ is -3.

---

2) For example, ‘‘Trapping or stopping (if supported) is disabled...’’ (F.8.2). Note that fetching a trap representation might perform a trap but is not required to (see 6.2.6.1).

4. Conformance

1 In this International Standard, ‘‘shall’’ is to be interpreted as a requirement on an implementation or on a program; conversely, ‘‘shall not’’ is to be interpreted as a prohibition.

2 If a ‘‘shall’’ or ‘‘shall not’’ requirement that appears outside of a constraint or runtime-constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this International Standard by the words ‘‘undefined behavior’’ or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe ‘‘behavior that is undefined’’.

3 A program that is correct in all other aspects, operating on correct data, containing unspecified behavior shall be a correct program and act in accordance with 5.1.2.3.

4 The implementation shall not successfully translate a preprocessing translation unit containing a **#error** preprocessing directive unless it is part of a group skipped by conditional inclusion.

5 A *strictly conforming program* shall use only those features of the language and library specified in this International Standard.[3] It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit.

6 The two forms of *conforming implementation* are hosted and freestanding. A *conforming hosted implementation* shall accept any strictly conforming program. A *conforming freestanding implementation* shall accept any strictly conforming program in which the use of the features specified in the library clause (clause 7) is confined to the contents of the standard headers **<float.h>**, **<iso646.h>**, **<limits.h>**, **<stdalign.h>**, **<stdarg.h>**, **<stdbool.h>**, **<stddef.h>**, **<stdint.h>**, and **<stdnoreturn.h>**. A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.[4]

---

3) A strictly conforming program can use conditional features (see 6.10.8.3) provided the use is guarded by an appropriate conditional inclusion preprocessing directive using the related macro. For example:

```
#ifdef __STDC_IEC_559__ /* FE_UPWARD defined */
   /* ... */
   fesetround(FE_UPWARD);
   /* ... */
#endif
```

4) This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this International Standard.

7 A *conforming program* is one that is acceptable to a conforming implementation.[5)]

8 An implementation shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions.

Forward references: conditional inclusion (6.10.1), error directive (6.10.5), characteristics of floating types **`<float.h>`** (7.7), alternative spellings **`<iso646.h>`** (7.9), sizes of integer types **`<limits.h>`** (7.10), alignment **`<stdalign.h>`** (7.15), variable arguments **`<stdarg.h>`** (7.16), boolean type and values **`<stdbool.h>`** (7.18), common definitions **`<stddef.h>`** (7.19), integer types **`<stdint.h>`** (7.20), **`<stdnoreturn.h>`** (7.23).

5) Strictly conforming programs are intended to be maximally portable among conforming implementations. Conforming programs may depend upon nonportable features of a conforming implementation.

5. Environment

1 An implementation translates C source files and executes C programs in two data-processing-system environments, which will be called the *translation environment* and the *execution environment* in this International Standard. Their characteristics define and constrain the results of executing conforming C programs constructed according to the syntactic and semantic rules for conforming implementations.

Forward references: In this clause, only a few of many possible forward references have been noted.

5.1 Conceptual models

5.1.1 Translation environment

5.1.1.1 Program structure

1 A C program need not all be translated at the same time. The text of the program is kept in units called *source files*, (or *preprocessing files*) in this International Standard. A source file together with all the headers and source files included via the preprocessing directive `#include` is known as a *preprocessing translation unit*. After preprocessing, a preprocessing translation unit is called a *translation unit*. Previously translated translation units may be preserved individually or in libraries. The separate translation units of a program communicate by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units may be separately translated and then later linked to produce an executable program.

Forward references: linkages of identifiers (6.2.2), external definitions (6.9), preprocessing directives (6.10).

5.1.1.2 Translation phases

1 The precedence among the syntax rules of translation is specified by the following phases.[6)]

1. Physical source file multibyte characters are mapped, in an implementation-defined manner, to the source character set (introducing new-line characters for end-of-line indicators) if necessary. Trigraph sequences are replaced by corresponding single-character internal representations.

---

6) Implementations shall behave as if these separate phases occur, even though many are typically folded together in practice. Source files, translation units, and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation.

2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character before any such splicing takes place.

3. The source file is decomposed into preprocessing tokens[7)] and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment. Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.

4. Preprocessing directives are executed, macro invocations are expanded, and **`_Pragma`** unary operator expressions are executed. If a character sequence that matches the syntax of a universal character name is produced by token concatenation (6.10.3.3), the behavior is undefined. A **`#include`** preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.

5. Each source character set member and escape sequence in character constants and string literals is converted to the corresponding member of the execution character set; if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.[8)]

6. Adjacent string literal tokens are concatenated.

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. The resulting tokens are syntactically and semantically analyzed and translated as a translation unit.

8. All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

Forward references: universal character names (6.4.3), lexical elements (6.4), preprocessing directives (6.10), trigraph sequences (5.2.1.1), external definitions (6.9).

---

7) As described in 6.4, the process of dividing a source file's characters into preprocessing tokens is context-dependent. For example, see the handling of **`<`** within a **`#include`** preprocessing directive.

8) An implementation need not convert all non-corresponding source characters to the same execution character.

5.1.1.3 Diagnostics

1 A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined. Diagnostic messages need not be produced in other circumstances.[9)]

2 EXAMPLE An implementation shall issue a diagnostic for the translation unit:

```
char i;
int i;
```

because in those cases where wording in this International Standard describes the behavior for a construct as being both a constraint error and resulting in undefined behavior, the constraint error shall be diagnosed.

5.1.2 Execution environments

1 Two execution environments are defined: *freestanding* and *hosted*. In both cases, *program startup* occurs when a designated C function is called by the execution environment. All objects with static storage duration shall be *initialized* (set to their initial values) before program startup. The manner and timing of such initialization are otherwise unspecified. *Program termination* returns control to the execution environment.

Forward references: storage durations of objects (6.2.4), initialization (6.7.9).

5.1.2.1 Freestanding environment

1 In a freestanding environment (in which C program execution may take place without any benefit of an operating system), the name and type of the function called at program startup are implementation-defined. Any library facilities available to a freestanding program, other than the minimal set required by clause 4, are implementation-defined.

2 The effect of program termination in a freestanding environment is implementation-defined.

5.1.2.2 Hosted environment

1 A hosted environment need not be provided, but shall conform to the following specifications if present.

9) The intent is that an implementation should identify the nature of, and where possible localize, each violation. Of course, an implementation is free to produce any number of diagnostics as long as a valid program is still correctly translated. It may also successfully translate an invalid program.

5.1.2.2.1 Program startup

1 The function called at program startup is named **`main`**. The implementation declares no prototype for this function. It shall be defined with a return type of **`int`** and with no parameters:

```
int main(void) { /* ... */ }
```

or with two parameters (referred to here as **`argc`** and **`argv`**, though any names may be used, as they are local to the function in which they are declared):

```
int main(int argc, char *argv[]) { /* ... */ }
```

or equivalent;[10)] or in some other implementation-defined manner.

2 If they are declared, the parameters to the **`main`** function shall obey the following constraints:

— The value of **`argc`** shall be nonnegative.

— **`argv[argc]`** shall be a null pointer.

— If the value of **`argc`** is greater than zero, the array members **`argv[0]`** through **`argv[argc-1]`** inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup. The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. If the host environment is not capable of supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase.

— If the value of **`argc`** is greater than zero, the string pointed to by **`argv[0]`** represents the *program name*; **`argv[0][0]`** shall be the null character if the program name is not available from the host environment. If the value of **`argc`** is greater than one, the strings pointed to by **`argv[1]`** through **`argv[argc-1]`** represent the *program parameters*.

— The parameters **`argc`** and **`argv`** and the strings pointed to by the **`argv`** array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.

5.1.2.2.2 Program execution

1 In a hosted environment, a program may use all the functions, macros, type definitions, and objects described in the library clause (clause 7).

---

10) Thus, **`int`** can be replaced by a typedef name defined as **`int`**, or the type of **`argv`** can be written as **`char ** argv`**, and so on.

5.1.2.2.3 Program termination

1 If the return type of the **`main`** function is a type compatible with **`int`**, a return from the initial call to the **`main`** function is equivalent to calling the **`exit`** function with the value returned by the **`main`** function as its argument;[11] reaching the **`}`** that terminates the **`main`** function returns a value of 0. If the return type is not compatible with **`int`**, the termination status returned to the host environment is unspecified.

Forward references: definition of terms (7.1.1), the **`exit`** function (7.22.4.4).

5.1.2.3 Program execution

1 The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.

2 Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*,[12] which are changes in the state of the execution environment. *Evaluation* of an expression in general includes both value computations and initiation of side effects. Value computation for an lvalue expression includes determining the identity of the designated object.

3 *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations *A* and *B*, if *A* is sequenced before *B*, then the execution of *A* shall precede the execution of *B*. (Conversely, if *A* is sequenced before *B*, then *B* is *sequenced after A*.) If *A* is not sequenced before or after *B*, then *A* and *B* are *unsequenced*. Evaluations *A* and *B* are *indeterminately sequenced* when *A* is sequenced either before or after *B*, but it is unspecified which.[13] The presence of a *sequence point* between the evaluation of expressions *A* and *B* implies that every value computation and side effect associated with *A* is sequenced before every value computation and side effect associated with *B*. (A summary of the sequence points is given in annex C.)

4 In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by

11) In accordance with 6.2.4, the lifetimes of objects with automatic storage duration declared in **`main`** will have ended in the former case, even where they would not have in the latter.

12) The IEC 60559 standard for binary floating-point arithmetic requires certain user-accessible status flags and control modes. Floating-point operations implicitly set the status flags; modes affect result values of floating-point operations. Implementations that support such floating-point state are required to regard changes to it as side effects — see annex F for details. The floating-point environment library **`<fenv.h>`** provides a programming facility for indicating when these side effects matter, freeing the implementations in other cases.

13) The executions of unsequenced evaluations can interleave. Indeterminately sequenced evaluations cannot interleave, but can be executed in any order.

calling a function or accessing a volatile object).

5 When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects that are neither lock-free atomic objects nor of type **volatile sig_atomic_t** are unspecified, as is the state of the floating-point environment. The value of any object modified by the handler that is neither a lock-free atomic object nor of type **volatile sig_atomic_t** becomes indeterminate when the handler exits, as does the state of the floating-point environment if it is modified by the handler and not restored to its original state.

6 The least requirements on a conforming implementation are:

— Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine.

— At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.

— The input and output dynamics of interactive devices shall take place as specified in 7.21.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input.

This is the *observable behavior* of the program.

7 What constitutes an interactive device is implementation-defined.

8 More stringent correspondences between abstract and actual semantics may be defined by each implementation.

9 EXAMPLE 1 An implementation might define a one-to-one correspondence between abstract and actual semantics: at every sequence point, the values of the actual objects would agree with those specified by the abstract semantics. The keyword **volatile** would then be redundant.

10 Alternatively, an implementation might perform various optimizations within each translation unit, such that the actual semantics would agree with the abstract semantics only when making function calls across translation unit boundaries. In such an implementation, at the time of each function entry and function return where the calling function and the called function are in different translation units, the values of all externally linked objects and of all objects accessible via pointers therein would agree with the abstract semantics. Furthermore, at the time of each such function entry the values of the parameters of the called function and of all objects accessible via pointers therein would agree with the abstract semantics. In this type of implementation, objects referred to by interrupt service routines activated by the **signal** function would require explicit specification of **volatile** storage, as well as other implementation-defined restrictions.

11 EXAMPLE 2 In executing the fragment

```
char c1, c2;
/* ... */
c1 = c1 + c2;
```

the ‘‘integer promotions’’ require that the abstract machine promote the value of each variable to **int** size and then add the two **int**s and truncate the sum. Provided the addition of two **char**s can be done without overflow, or with overflow wrapping silently to produce the correct result, the actual execution need only produce the same result, possibly omitting the promotions.

12 EXAMPLE 3 Similarly, in the fragment

```
float f1, f2;
double d;
/* ... */
f1 = f2 * d;
```

the multiplication may be executed using single-precision arithmetic if the implementation can ascertain that the result would be the same as if it were executed using double-precision arithmetic (for example, if **`d`** were replaced by the constant **`2.0`**, which has type **`double`**).

13 EXAMPLE 4 Implementations employing wide registers have to take care to honor appropriate semantics. Values are independent of whether they are represented in a register or in memory. For example, an implicit *spilling* of a register is not permitted to alter the value. Also, an explicit *store and load* is required to round to the precision of the storage type. In particular, casts and assignments are required to perform their specified conversion. For the fragment

```
double d1, d2;
float f;
d1 = f = expression;
d2 = (float) expression;
```

the values assigned to **`d1`** and **`d2`** are required to have been converted to **`float`**.

14 EXAMPLE 5 Rearrangement for floating-point expressions is often restricted because of limitations in precision as well as range. The implementation cannot generally apply the mathematical associative rules for addition or multiplication, nor the distributive rule, because of roundoff error, even in the absence of overflow and underflow. Likewise, implementations cannot generally replace decimal constants in order to rearrange expressions. In the following fragment, rearrangements suggested by mathematical rules for real numbers are often not valid (see F.9).

```
double x, y, z;
/* ... */
x = (x * y) * z;   // not equivalent to x *= y * z;
z = (x - y) + y ;  // not equivalent to z = x;
z = x + x * y;     // not equivalent to z = x * (1.0 + y);
y = x / 5.0;       // not equivalent to y = x * 0.2;
```

15 EXAMPLE 6 To illustrate the grouping behavior of expressions, in the following fragment

```
int a, b;
/* ... */
a = a + 32760 + b + 5;
```

the expression statement behaves exactly the same as

```
a = (((a + 32760) + b) + 5);
```

due to the associativity and precedence of these operators. Thus, the result of the sum **`(a + 32760)`** is next added to **`b`**, and that result is then added to **`5`** which results in the value assigned to **`a`**. On a machine in which overflows produce an explicit trap and in which the range of values representable by an **`int`** is [−32768, +32767], the implementation cannot rewrite this expression as

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

since if the values for **`a`** and **`b`** were, respectively, −32754 and −15, the sum **`a + b`** would produce a trap while the original expression would not; nor can the expression be rewritten either as

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

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 standard. 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 may 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.

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.

5 The library defines a number of *atomic operations* (7.17) and operations on mutexes (7.26.4) that are specially identified as synchronization operations. These operations play a special role in making assignments 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, and atomic *read-modify-write operations*, which have special characteristics.

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 that later perform an acquire or consume operation on *A*. We do not include relaxed atomic operations 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 must 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 may 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 *synchronize with* other library calls 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:

15) The ‘‘carries a dependency’’ relation is a subset of the ‘‘sequenced before’’ relation, and is similarly strictly intra-thread.

— 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*.

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

18 An evaluation *A happens before* an evaluation *B* if *A* is sequenced before *B* or *A* inter-thread happens before *B*.

16) The ‘‘dependency-ordered before’’ relation is analogous to the ‘‘synchronizes with’’ relation, but uses release/consume in place of release/acquire.

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

— A happens before B, and

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

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

20 NOTE 8 If there is ambiguity about which side effect to a non-atomic object is visible, then there is a data race and the behavior is undefined.

21 NOTE 9 This states that operations on ordinary variables are not visibly reordered. This is not actually detectable without data races, but it is necessary to ensure that data races, as defined here, and with suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution.

22 The *visible sequence of side effects* on an atomic object M, with respect to a value computation B of M, is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first side effect is visible with respect to B, and for every subsequent side effect, it is not the case that B happens before it. The value of an atomic object M, as determined by evaluation B, shall be the value stored by some operation in the visible sequence of M with respect to B. Furthermore, if a value computation A of an atomic object M happens before a value computation B of M, and the value computed by A corresponds to the value stored by side effect X, then the value computed by B shall either equal the value computed by A, or be the value stored by side effect Y, where Y follows X in the modification order of M.

23 NOTE 10 This effectively disallows compiler reordering of atomic operations to a single object, even if both operations are ‘‘relaxed’’ loads. By doing so, we effectively make the ‘‘cache coherence’’ guarantee provided by most hardware available to C atomic operations.

24 NOTE 11 The visible sequence depends on the ‘‘happens before’’ relation, which in turn depends on the values observed by loads of atomics, which we are restricting here. The intended reading is that there must exist an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the ‘‘happens before’’ relation derived as described above, satisfy the resulting constraints as imposed here.

25 The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

26 NOTE 12 It can be shown that programs that correctly use simple mutexes and **`memory_order_seq_cst`** operations to prevent all data races, and use no other synchronization operations, behave as though the operations executed by their constituent threads were simply interleaved, with each value computation of an object being the last value stored in that interleaving. This is normally referred to as ‘‘sequential consistency’’. However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result must contain undefined behavior.

27 NOTE 13 Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this standard, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. We also generally preclude reordering of atomic loads in cases in which the atomics in question may alias, since this may violate the "visible sequence" rules.

28 NOTE 14 Transformations that introduce a speculative read of a potentially shared memory location may not preserve the semantics of the program as defined in this standard, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection.

5.2 Environmental considerations

5.2.1 Character sets

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

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

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

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

the 26 *lowercase letters* of the Latin alphabet

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

the 10 decimal *digits*

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

the following 29 graphic characters

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

the space character, and control characters representing horizontal tab, vertical tab, and form feed. The representation of each member of the source and execution basic character sets shall fit in a byte. In both the source and execution basic character sets, the value of each character after `0` in the above list of decimal digits shall be one greater than the value of the previous. In source files, there shall be some way of indicating the end of each line of text; this International Standard treats such an end-of-line indicator as if it were a single new-line character. In the basic execution character set, there shall be control characters representing alert, backspace, carriage return, and new line. If any other characters are encountered in a source file (except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token), the behavior is undefined.

4 A *letter* is an uppercase letter or a lowercase letter as defined above; in this International Standard the term does not include other characters that are letters in other alphabets.

5 The universal character name construct provides a way to name other characters.

Forward references: universal character names (6.4.3), character constants (6.4.4.4), preprocessing directives (6.10), string literals (6.4.5), comments (6.4.9), string (7.1.1).

5.2.1.1 Trigraph sequences

1 Before any other processing takes place, each occurrence of one of the following sequences of three characters (called *trigraph sequences*[17]) is replaced with the corresponding single character.

```
??=  #          ??)  ]          ??!  |
??(  [          ??'  ^          ??>  }
??/  \          ??<  {          ??-  ~
```

No other trigraph sequences exist. Each **?** that does not begin one of the trigraphs listed above is not changed.

2 EXAMPLE 1

```
??=define arraycheck(a, b) a??(b??) ??!??! b??(a??)
```

becomes

```
#define arraycheck(a, b) a[b] || b[a]
```

3 EXAMPLE 2 The following source line

```
printf("Eh???/n");
```

becomes (after replacement of the trigraph sequence **??/**)

```
printf("Eh?\n");
```

5.2.1.2 Multibyte characters

1 The source character set may contain multibyte characters, used to represent members of the extended character set. The execution character set may also contain multibyte characters, which need not have the same encoding as for the source character set. For both character sets, the following shall hold:

— The basic character set shall be present and each character shall be encoded as a single byte.

— The presence, meaning, and representation of any additional members is locale-specific.

17) The trigraph sequences enable the input of characters that are not defined in the Invariant Code Set as described in ISO/IEC 646, which is a subset of the seven-bit US ASCII code set.

— A multibyte character set may have a *state-dependent encoding*, wherein each sequence of multibyte characters begins in an *initial shift state* and enters other locale-specific *shift states* when specific multibyte characters are encountered in the sequence. While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes in the sequence is a function of the current shift state.

— A byte with all bits zero shall be interpreted as a null character independent of shift state. Such a byte shall not occur as part of any other multibyte character.

2 For source files, the following shall hold:

— An identifier, comment, string literal, character constant, or header name shall begin and end in the initial shift state.

— An identifier, comment, string literal, character constant, or header name shall consist of a sequence of valid multibyte characters.

5.2.2 Character display semantics

1 The *active position* is that location on a display device where the next character output by the **`fputc`** function would appear. The intent of writing a printing character (as defined by the **`isprint`** function) to a display device is to display a graphic representation of that character at the active position and then advance the active position to the next position on the current line. The direction of writing is locale-specific. If the active position is at the final position of a line (if there is one), the behavior of the display device is unspecified.

2 Alphabetic escape sequences representing nongraphic characters in the execution character set are intended to produce actions on display devices as follows:

`\a` (*alert*) Produces an audible or visible alert without changing the active position.

`\b` (*backspace*) Moves the active position to the previous position on the current line. If the active position is at the initial position of a line, the behavior of the display device is unspecified.

`\f` (*form feed*) Moves the active position to the initial position at the start of the next logical page.

`\n` (*new line*) Moves the active position to the initial position of the next line.

`\r` (*carriage return*) Moves the active position to the initial position of the current line.

`\t` (*horizontal tab*) Moves the active position to the next horizontal tabulation position on the current line. If the active position is at or past the last defined horizontal tabulation position, the behavior of the display device is unspecified.

`\v` (*vertical tab*) Moves the active position to the initial position of the next vertical tabulation position. If the active position is at or past the last defined vertical tabulation position, the behavior of the display device is unspecified.

3 Each of these escape sequences shall produce a unique implementation-defined value which can be stored in a single **char** object. The external representations in a text file need not be identical to the internal representations, and are outside the scope of this International Standard.

Forward references: the **isprint** function (7.4.1.8), the **fputc** function (7.21.7.3).

5.2.3 Signals and interrupts

1 Functions shall be implemented such that they may be interrupted at any time by a signal, or may be called by a signal handler, or both, with no alteration to earlier, but still active, invocations' control flow (after the interruption), function return values, or objects with automatic storage duration. All such objects shall be maintained outside the *function image* (the instructions that compose the executable representation of a function) on a per-invocation basis.

5.2.4 Environmental limits

1 Both the translation and execution environments constrain the implementation of language translators and libraries. The following summarizes the language-related environmental limits on a conforming implementation; the library-related limits are discussed in clause 7.

5.2.4.1 Translation limits

1 The implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits:[18)]

— 127 nesting levels of blocks

— 63 nesting levels of conditional inclusion

— 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or **void** type in a declaration

— 63 nesting levels of parenthesized declarators within a full declarator

— 63 nesting levels of parenthesized expressions within a full expression

— 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character)

— 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each

18) Implementations should avoid imposing fixed translation limits whenever possible.

universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)[19]

— 4095 external identifiers in one translation unit

— 511 identifiers with block scope declared in one block

— 4095 macro identifiers simultaneously defined in one preprocessing translation unit

— 127 parameters in one function definition

— 127 arguments in one function call

— 127 parameters in one macro definition

— 127 arguments in one macro invocation

— 4095 characters in a logical source line

— 4095 characters in a string literal (after concatenation)

— 65535 bytes in an object (in a hosted environment only)

— 15 nesting levels for **#include**d files

— 1023 **case** labels for a **switch** statement (excluding those for any nested **switch** statements)

— 1023 members in a single structure or union

— 1023 enumeration constants in a single enumeration

— 63 levels of nested structure or union definitions in a single struct-declaration-list

5.2.4.2 Numerical limits

1 An implementation is required to document all the limits specified in this subclause, which are specified in the headers **<limits.h>** and **<float.h>**. Additional limits are specified in **<stdint.h>**.

Forward references: integer types **<stdint.h>** (7.20).

5.2.4.2.1 Sizes of integer types <limits.h>

1 The values given below shall be replaced by constant expressions suitable for use in **#if** preprocessing directives. Moreover, except for **CHAR_BIT** and **MB_LEN_MAX**, the following shall be replaced by expressions that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Their implementation-defined values shall be equal or greater in magnitude

19) See ‘‘future language directions’’ (6.11.3).

(absolute value) to those shown, with the same sign.

— number of bits for smallest object that is not a bit-field (byte)
`CHAR_BIT` **`8`**

— minimum value for an object of type **`signed char`**
`SCHAR_MIN` **`-127 //`** $-(2^7 - 1)$

— maximum value for an object of type **`signed char`**
`SCHAR_MAX` **`+127 //`** $2^7 - 1$

— maximum value for an object of type **`unsigned char`**
`UCHAR_MAX` **`255 //`** $2^8 - 1$

— minimum value for an object of type **`char`**
`CHAR_MIN` *see below*

— maximum value for an object of type **`char`**
`CHAR_MAX` *see below*

— maximum number of bytes in a multibyte character, for any supported locale
`MB_LEN_MAX` **`1`**

— minimum value for an object of type **`short int`**
`SHRT_MIN` **`-32767 //`** $-(2^{15} - 1)$

— maximum value for an object of type **`short int`**
`SHRT_MAX` **`+32767 //`** $2^{15} - 1$

— maximum value for an object of type **`unsigned short int`**
`USHRT_MAX` **`65535 //`** $2^{16} - 1$

— minimum value for an object of type **`int`**
`INT_MIN` **`-32767 //`** $-(2^{15} - 1)$

— maximum value for an object of type **`int`**
`INT_MAX` **`+32767 //`** $2^{15} - 1$

— maximum value for an object of type **`unsigned int`**
`UINT_MAX` **`65535 //`** $2^{16} - 1$

— minimum value for an object of type **`long int`**
`LONG_MIN` **`-2147483647 //`** $-(2^{31} - 1)$

— maximum value for an object of type **`long int`**
`LONG_MAX` **`+2147483647 //`** $2^{31} - 1$

— maximum value for an object of type **`unsigned long int`**
`ULONG_MAX` **`4294967295 //`** $2^{32} - 1$

— minimum value for an object of type **`long long int`**
`LLONG_MIN` **`-9223372036854775807 //`** $-(2^{63}-1)$

— maximum value for an object of type **`long long int`**
`LLONG_MAX` **`+9223372036854775807 //`** $2^{63}-1$

— maximum value for an object of type **`unsigned long long int`**
`ULLONG_MAX` **`18446744073709551615 //`** $2^{64}-1$

2 If the value of an object of type **`char`** is treated as a signed integer when used in an expression, the value of **`CHAR_MIN`** shall be the same as that of **`SCHAR_MIN`** and the value of **`CHAR_MAX`** shall be the same as that of **`SCHAR_MAX`**. Otherwise, the value of **`CHAR_MIN`** shall be 0 and the value of **`CHAR_MAX`** shall be the same as that of **`UCHAR_MAX`**.[20] The value **`UCHAR_MAX`** shall equal $2^{\texttt{CHAR\_BIT}}-1$.

Forward references: representations of types (6.2.6), conditional inclusion (6.10.1).

5.2.4.2.2 Characteristics of floating types `<float.h>`

1 The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.[21] The following parameters are used to define the model for each floating-point type:

s	sign (±1)
b	base or radix of exponent representation (an integer > 1)
e	exponent (an integer between a minimum $e_{\min}$ and a maximum $e_{\max}$)
p	precision (the number of base-b digits in the significand)
f_k	nonnegative integers less than b (the significand digits)

2 A *floating-point number* (x) is defined by the following model:

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{\min} \le e \le e_{\max}$$

3 In addition to normalized floating-point numbers ($f_1 > 0$ if $x \ne 0$), floating types may be able to contain other kinds of floating-point numbers, such as *subnormal floating-point numbers* ($x \ne 0$, $e = e_{\min}$, $f_1 = 0$) and *unnormalized floating-point numbers* ($x \ne 0$, $e > e_{\min}$, $f_1 = 0$), and values that are not floating-point numbers, such as infinities and NaNs. A *NaN* is an encoding signifying Not-a-Number. A *quiet NaN* propagates through almost every arithmetic operation without raising a floating-point exception; a *signaling NaN* generally raises a floating-point exception when occurring as an

20) See 6.2.5.

21) The floating-point model is intended to clarify the description of each floating-point characteristic and does not require the floating-point arithmetic of the implementation to be identical.

arithmetic operand.[22)]

4 An implementation may give zero and values that are not floating-point numbers (such as infinities and NaNs) a sign or may leave them unsigned. Wherever such values are unsigned, any requirement in this International Standard to retrieve the sign shall produce an unspecified sign, and any requirement to set the sign shall be ignored.

5 The minimum range of representable values for a floating type is the most negative finite floating-point number representable in that type through the most positive finite floating-point number representable in that type. In addition, if negative infinity is representable in a type, the range of that type is extended to all negative real numbers; likewise, if positive infinity is representable in a type, the range of that type is extended to all positive real numbers.

6 The accuracy of the floating-point operations (`+`, `-`, `*`, `/`) and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results is implementation-defined, as is the accuracy of the conversion between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>`. The implementation may state that the accuracy is unknown.

7 All integer values in the `<float.h>` header, except `FLT_ROUNDS`, shall be constant expressions suitable for use in `#if` preprocessing directives; all floating values shall be constant expressions. All except `DECIMAL_DIG`, `FLT_EVAL_METHOD`, `FLT_RADIX`, and `FLT_ROUNDS` have separate names for all three floating-point types. The floating-point model representation is provided for all values except `FLT_EVAL_METHOD` and `FLT_ROUNDS`.

8 The rounding mode for floating-point addition is characterized by the implementation-defined value of `FLT_ROUNDS`:[23)]

`-1`	indeterminable
`0`	toward zero
`1`	to nearest
`2`	toward positive infinity
`3`	toward negative infinity

All other values for `FLT_ROUNDS` characterize implementation-defined rounding behavior.

---

22) IEC 60559:1989 specifies quiet and signaling NaNs. For implementations that do not support IEC 60559:1989, the terms quiet NaN and signaling NaN are intended to apply to encodings with similar behavior.

23) Evaluation of `FLT_ROUNDS` correctly reflects any execution-time change of rounding mode through the function `fesetround` in `<fenv.h>`.

9 Except for assignment and cast (which remove all extra range and precision), the values yielded by operators with floating operands and values subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type. The use of evaluation formats is characterized by the implementation-defined value of **`FLT_EVAL_METHOD`**:[24)]

- **`-1`** indeterminable;
- **`0`** evaluate all operations and constants just to the range and precision of the type;
- **`1`** evaluate operations and constants of type **`float`** and **`double`** to the range and precision of the **`double`** type, evaluate **`long double`** operations and constants to the range and precision of the **`long double`** type;
- **`2`** evaluate all operations and constants to the range and precision of the **`long double`** type.

All other negative values for **`FLT_EVAL_METHOD`** characterize implementation-defined behavior.

10 The presence or absence of subnormal numbers is characterized by the implementation-defined values of **`FLT_HAS_SUBNORM`**, **`DBL_HAS_SUBNORM`**, and **`LDBL_HAS_SUBNORM`**:

- **`-1`** indeterminable[25)]
- **`0`** absent[26)] (type does not support subnormal numbers)
- **`1`** present (type does support subnormal numbers)

11 The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater or equal in magnitude (absolute value) to those shown, with the same sign:

— radix of exponent representation, *b*
`FLT_RADIX` **`2`**

24) The evaluation method determines evaluation formats of expressions involving all floating types, not just real types. For example, if **`FLT_EVAL_METHOD`** is 1, then the product of two **`float _Complex`** operands is represented in the **`double _Complex`** format, and its parts are evaluated to **`double`**.

25) Characterization as indeterminable is intended if floating-point operations do not consistently interpret subnormal representations as zero, nor as nonzero.

26) Characterization as absent is intended if no floating-point operations produce subnormal results from non-subnormal inputs, even if the type format includes representations of subnormal numbers.

— number of base-**`FLT_RADIX`** digits in the floating-point significand, p

`FLT_MANT_DIG`
`DBL_MANT_DIG`
`LDBL_MANT_DIG`

— number of decimal digits, n, such that any floating-point number with p radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value,

$$\begin{cases} p \log_{10} b & \text{if } b \text{ is a power of 10} \\ \lceil 1 + p \log_{10} b \rceil & \text{otherwise} \end{cases}$$

```
FLT_DECIMAL_DIG                  6
DBL_DECIMAL_DIG                 10
LDBL_DECIMAL_DIG                10
```

— number of decimal digits, n, such that any floating-point number in the widest supported floating type with p_{max} radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value,

$$\begin{cases} p_{\max} \log_{10} b & \text{if } b \text{ is a power of 10} \\ \lceil 1 + p_{\max} \log_{10} b \rceil & \text{otherwise} \end{cases}$$

```
DECIMAL_DIG                     10
```

— number of decimal digits, q, such that any floating-point number with q decimal digits can be rounded into a floating-point number with p radix b digits and back again without change to the q decimal digits,

$$\begin{cases} p \log_{10} b & \text{if } b \text{ is a power of 10} \\ \lfloor (p-1) \log_{10} b \rfloor & \text{otherwise} \end{cases}$$

```
FLT_DIG                          6
DBL_DIG                         10
LDBL_DIG                        10
```

— minimum negative integer such that **`FLT_RADIX`** raised to one less than that power is a normalized floating-point number, e_{min}

`FLT_MIN_EXP`
`DBL_MIN_EXP`
`LDBL_MIN_EXP`

— minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\left\lceil \log_{10} b^{e_{\min}-1} \right\rceil$

```
FLT_MIN_10_EXP                   -37
DBL_MIN_10_EXP                   -37
LDBL_MIN_10_EXP                  -37
```

— maximum integer such that **FLT_RADIX** raised to one less than that power is a representable finite floating-point number, $e_{\max}$

```
FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP
```

— maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\lfloor \log_{10}((1 - b^{-p})b^{e_{\max}}) \rfloor$

```
FLT_MAX_10_EXP                   +37
DBL_MAX_10_EXP                   +37
LDBL_MAX_10_EXP                  +37
```

12 The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater than or equal to those shown:

— maximum representable finite floating-point number, $(1 - b^{-p})b^{e_{\max}}$

```
FLT_MAX                        1E+37
DBL_MAX                        1E+37
LDBL_MAX                       1E+37
```

13 The values given in the following list shall be replaced by constant expressions with implementation-defined (positive) values that are less than or equal to those shown:

— the difference between 1 and the least value greater than 1 that is representable in the given floating point type, b^{1-p}

```
FLT_EPSILON                     1E-5
DBL_EPSILON                     1E-9
LDBL_EPSILON                    1E-9
```

— minimum normalized positive floating-point number, $b^{e_{\min}-1}$

```
FLT_MIN                        1E-37
DBL_MIN                        1E-37
LDBL_MIN                       1E-37
```

— minimum positive floating-point number[27)]

```
FLT_TRUE_MIN                   1E-37
DBL_TRUE_MIN                   1E-37
LDBL_TRUE_MIN                  1E-37
```

Recommended practice

14 Conversion from (at least) **`double`** to decimal with **`DECIMAL_DIG`** digits and back should be the identity function.

15 EXAMPLE 1 The following describes an artificial floating-point representation that meets the minimum requirements of this International Standard, and the appropriate values in a **`<float.h>`** header for type **`float`**:

$$x = s16^e \sum_{k=1}^{6} f_k 16^{-k}, \quad -31 \le e \le +32$$

```
FLT_RADIX                          16
FLT_MANT_DIG                        6
FLT_EPSILON            9.53674316E-07F
FLT_DECIMAL_DIG                     9
FLT_DIG                             6
FLT_MIN_EXP                       -31
FLT_MIN                2.93873588E-39F
FLT_MIN_10_EXP                    -38
FLT_MAX_EXP                       +32
FLT_MAX                3.40282347E+38F
FLT_MAX_10_EXP                    +38
```

16 EXAMPLE 2 The following describes floating-point representations that also meet the requirements for single-precision and double-precision numbers in IEC 60559,[28)] and the appropriate values in a **`<float.h>`** header for types **`float`** and **`double`**:

$$x_f = s2^e \sum_{k=1}^{24} f_k 2^{-k}, \quad -125 \le e \le +128$$

$$x_d = s2^e \sum_{k=1}^{53} f_k 2^{-k}, \quad -1021 \le e \le +1024$$

```
FLT_RADIX                           2
DECIMAL_DIG                        17
FLT_MANT_DIG                       24
FLT_EPSILON            1.19209290E-07F // decimal constant
FLT_EPSILON                 0X1P-23F // hex constant
FLT_DECIMAL_DIG                     9
```

---

27) If the presence or absence of subnormal numbers is indeterminable, then the value is intended to be a positive number no greater than the minimum normalized positive number for the type.

28) The floating-point model in that standard sums powers of *b* from zero, so the values of the exponent limits are one less than shown here.

```
FLT_DIG                            6
FLT_MIN_EXP                     -125
FLT_MIN              1.17549435E-38F // decimal constant
FLT_MIN                   0X1P-126F  // hex constant
FLT_TRUE_MIN         1.40129846E-45F // decimal constant
FLT_TRUE_MIN              0X1P-149F  // hex constant
FLT_HAS_SUBNORM                    1
FLT_MIN_10_EXP                   -37
FLT_MAX_EXP                     +128
FLT_MAX              3.40282347E+38F // decimal constant
FLT_MAX              0X1.fffffeP127F // hex constant
FLT_MAX_10_EXP                   +38
DBL_MANT_DIG                      53
DBL_EPSILON   2.2204460492503131E-16 // decimal constant
DBL_EPSILON                 0X1P-52  // hex constant
DBL_DECIMAL_DIG                   17
DBL_DIG                           15
DBL_MIN_EXP                    -1021
DBL_MIN      2.2250738585072014E-308 // decimal constant
DBL_MIN                   0X1P-1022  // hex constant
DBL_TRUE_MIN 4.9406564584124654E-324 // decimal constant
DBL_TRUE_MIN              0X1P-1074  // hex constant
DBL_HAS_SUBNORM                    1
DBL_MIN_10_EXP                  -307
DBL_MAX_EXP                    +1024
DBL_MAX      1.7976931348623157E+308 // decimal constant
DBL_MAX       0X1.fffffffffffffP1023 // hex constant
DBL_MAX_10_EXP                  +308
```

If a type wider than **double** were supported, then **DECIMAL_DIG** would be greater than 17. For example, if the widest type were to use the minimal-width IEC 60559 double-extended format (64 bits of precision), then **DECIMAL_DIG** would be 21.

Forward references: conditional inclusion (6.10.1), complex arithmetic **<complex.h>** (7.3), extended multibyte and wide character utilities **<wchar.h>** (7.29), floating-point environment **<fenv.h>** (7.6), general utilities **<stdlib.h>** (7.22), input/output **<stdio.h>** (7.21), mathematics **<math.h>** (7.12).

6. Language

6.1 Notation

1 In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by *italic type*, and literal words and character set members (terminals) by **bold type**. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words ‘‘one of’’. An optional symbol is indicated by the subscript ‘‘opt’’, so that

{ $expression_{opt}$ **}**

indicates an optional expression enclosed in braces.

2 When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

3 A summary of the language syntax is given in annex A.

6.2 Concepts

6.2.1 Scopes of identifiers

1 An identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. The same identifier can denote different entities at different points in the program. A member of an enumeration is called an *enumeration constant*. Macro names and macro parameters are not considered further here, because prior to the semantic phase of program translation any occurrences of macro names in the source file are replaced by the preprocessing token sequences that constitute their macro definitions.

2 For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. Different entities designated by the same identifier either have different scopes, or are in different name spaces. There are four kinds of scopes: function, file, block, and function prototype. (A *function prototype* is a declaration of a function that declares the types of its parameters.)

3 A label name is the only kind of identifier that has *function scope*. It can be used (in a **goto** statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance (followed by a **:** and a statement).

4 Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit. If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the end of the associated block. If the declarator or type specifier that declares the identifier appears

within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator. If an identifier designates two different entities in the same name space, the scopes might overlap. If so, the scope of one entity (the *inner scope*) will end strictly before the scope of the other entity (the *outer scope*). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is *hidden* (and not visible) within the inner scope.

5 Unless explicitly stated otherwise, where this International Standard uses the term ‘‘identifier’’ to refer to some entity (as opposed to the syntactic construct), it refers to the entity in the relevant name space whose declaration is visible at the point the identifier occurs.

6 Two identifiers have the *same scope* if and only if their scopes terminate at the same point.

7 Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. Any other identifier has scope that begins just after the completion of its declarator.

8 As a special case, a type name (which is not a declaration of an identifier) is considered to have a scope that begins just after the place within the type name where the omitted identifier would appear were it not omitted.

Forward references: declarations (6.7), function calls (6.5.2.2), function definitions (6.9.1), identifiers (6.4.2), macro replacement (6.10.3), name spaces of identifiers (6.2.3), source file inclusion (6.10.2), statements (6.8).

6.2.2 Linkages of identifiers

1 An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*.[29] There are three kinds of linkage: external, internal, and none.

2 In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with *external linkage* denotes the same object or function. Within one translation unit, each declaration of an identifier with *internal linkage* denotes the same object or function. Each declaration of an identifier with *no linkage* denotes a unique entity.

3 If the declaration of a file scope identifier for an object or a function contains the storage-class specifier **`static`**, the identifier has internal linkage.[30]

29) There is no linkage between different identifiers.

4 For an identifier declared with the storage-class specifier `extern` in a scope in which a prior declaration of that identifier is visible,[31] if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

5 If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier `extern`. If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.

6 The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier `extern`.

7 If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

Forward references: declarations (6.7), expressions (6.5), external definitions (6.9), statements (6.8).

6.2.3 Name spaces of identifiers

1 If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate *name spaces* for various categories of identifiers, as follows:

— *label names* (disambiguated by the syntax of the label declaration and use);

— the *tags* of structures, unions, and enumerations (disambiguated by following any[32] of the keywords `struct`, `union`, or `enum`);

— the *members* of structures or unions; each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the `.` or `->` operator);

— all other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants).

Forward references: enumeration specifiers (6.7.2.2), labeled statements (6.8.1), structure and union specifiers (6.7.2.1), structure and union members (6.5.2.3), tags (6.7.2.3), the `goto` statement (6.8.6.1).

---

30) A function declaration can contain the storage-class specifier `static` only if it is at file scope; see 6.7.1.

31) As specified in 6.2.1, the later declaration might hide the prior declaration.

32) There is only one name space for tags even though three are possible.

6.2.4 Storage durations of objects

1 An object has a *storage duration* that determines its lifetime. There are four storage durations: static, thread, automatic, and allocated. Allocated storage is described in 7.22.3.

2 The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address,[33)] and retains its last-stored value throughout its lifetime.[34)] If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.

3 An object whose identifier is declared without the storage-class specifier **`_Thread_local`**, and either with external or internal linkage or with the storage-class specifier **`static`**, has *static storage duration*. Its lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

4 An object whose identifier is declared with the storage-class specifier **`_Thread_local`** has *thread storage duration*. Its lifetime is the entire execution of the thread for which it is created, and its stored value is initialized when the thread is started. There is a distinct object per thread, and use of the declared name in an expression refers to the object associated with the thread evaluating the expression. The result of attempting to indirectly access an object with thread storage duration from a thread other than the one with which the object is associated is implementation-defined.

5 An object whose identifier is declared with no linkage and without the storage-class specifier **`static`** has *automatic storage duration*, as do some compound literals. The result of attempting to indirectly access an object with automatic storage duration from a thread other than the one with which the object is associated is implementation-defined.

6 For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.) If the block is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate. If an initialization is specified for the object, it is performed each time the declaration or compound literal is reached in the execution of the block; otherwise, the value becomes indeterminate each time the declaration is reached.

---

33) The term ‘‘constant address’’ means that two pointers to the object constructed at possibly different times will compare equal. The address may be different during two different executions of the same program.

34) In the case of a volatile object, the last store need not be explicit in the program.

7 For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.[35] If the scope is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate.

8 A non-lvalue expression with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all contained structures and unions) refers to an object with automatic storage duration and *temporary lifetime*.[36] Its lifetime begins when the expression is evaluated and its initial value is the value of the expression. Its lifetime ends when the evaluation of the containing full expression or full declarator ends. Any attempt to modify an object with temporary lifetime results in undefined behavior.

Forward references: array declarators (6.7.6.2), compound literals (6.5.2.5), declarators (6.7.6), function calls (6.5.2.2), initialization (6.7.9), statements (6.8).

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).[37]

2 An object declared as type **`_Bool`** is large enough to store the values 0 and 1.

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*.[38] The standard and extended signed integer types are collectively called *signed integer types*.[39]

---

35) Leaving the innermost block containing the declaration, or jumping to a point in that block or an embedded block prior to the declaration, leaves the scope of the declaration.

36) The address of such an object is taken implicitly when an array member is accessed.

37) A type may be incomplete or complete throughout an entire translation unit, or it may change states at different points within a translation unit.

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*.[40)]

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.[41)] 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 *real floating types*, designated as **float**, **double**, and **long double**.[42)] 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**.

---

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

39) Therefore, any statement in this Standard about signed integer types also applies to the extended signed integer types.

40) Therefore, any statement in this Standard about unsigned integer types also applies to the extended unsigned integer types.

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

42) See ''future language directions'' (6.11.1).

11 There are three *complex types*, designated as **`float _Complex`**, **`double _Complex`**, and **`long double _Complex`**.[43] (Complex types are a conditional feature that implementations need not support; see 6.10.8.3.) The real floating and complex types are collectively called the *floating types*.

12 For each floating type there is a *corresponding real type*, which is always a real floating type. For real floating types, it is the same type. For complex types, it is the type given by deleting the keyword **`_Complex`** from the type name.

13 Each complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; the first element is equal to the real part, and the second element to the imaginary part, of the complex number.

14 The type **`char`**, the signed and unsigned integer types, and the floating types are collectively called the *basic types*. The basic types are complete object types. Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.[44]

15 The three types **`char`**, **`signed char`**, and **`unsigned char`** are collectively called the *character types*. The implementation shall define **`char`** to have the same range, representation, and behavior as either **`signed char`** or **`unsigned char`**.[45]

16 An *enumeration* comprises a set of named integer constant values. Each distinct enumeration constitutes a different *enumerated type*.

17 The type **`char`**, the signed and unsigned integer types, and the enumerated types are collectively called *integer types*. The integer and real floating types are collectively called *real types*.

18 Integer and floating types are collectively called *arithmetic types*. Each arithmetic type belongs to one *type domain*: the *real type domain* comprises the real types, the *complex type domain* comprises the complex types.

19 The **`void`** type comprises an empty set of values; it is an incomplete object type that cannot be completed.

---

43) A specification for imaginary types is in annex G.

44) An implementation may define new keywords that provide alternative ways to designate a basic (or any other) type; this does not violate the requirement that all basic types be different. Implementation-defined keywords shall have the form of an identifier reserved for any use as described in 7.1.3.

45) **`CHAR_MIN`**, defined in **`<limits.h>`**, will have one of the values 0 or **`SCHAR_MIN`**, and this can be used to distinguish the two options. Irrespective of the choice made, **`char`** is a separate type from the other two and is not compatible with either.

20 Any number of *derived types* can be constructed from the object and function types, as follows:

— An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*. The element type shall be complete whenever the array type is specified. Array types are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is *T*, the array type is sometimes called ''array of *T*''. The construction of an array type from an element type is called ''array type derivation''.

— A *structure type* describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.

— A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

— A *function type* describes a function with specified return type. A function type is characterized by its return type and the number and types of its parameters. A function type is said to be derived from its return type, and if its return type is *T*, the function type is sometimes called ''function returning *T*''. The construction of a function type from a return type is called ''function type derivation''.

— A *pointer type* may be derived from a function type or an object type, called the *referenced type*. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type *T* is sometimes called ''pointer to *T*''. The construction of a pointer type from a referenced type is called ''pointer type derivation''. A pointer type is a complete object type.

— An *atomic type* describes the type designated by the construct **`_Atomic (`** *type-name* **`).`** (Atomic types are a conditional feature that implementations need not support; see 6.10.8.3.)

These methods of constructing derived types can be applied recursively.

21 Arithmetic types and pointer types are collectively called *scalar types*. Array and structure types are collectively called *aggregate types*.[46)]

22 An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure or union type of unknown content (as described in 6.7.2.3) is an incomplete

46) Note that aggregate type does not include union type because an object with union type can only contain one member at a time.

type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

23 A type has *known constant size* if the type is not incomplete and is not a variable length array type.

24 Array, function, and pointer types are collectively called *derived declarator types*. A *declarator type derivation* from a type *T* is the construction of a derived declarator type from *T* by the application of an array-type, a function-type, or a pointer-type derivation to *T*.

25 A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types.

26 Any type so far mentioned is an *unqualified type*. Each unqualified type has several *qualified versions* of its type,[47] corresponding to the combinations of one, two, or all three of the **`const`**, **`volatile`**, and **`restrict`** qualifiers. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[48] A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

27 Further, there is the **`_Atomic`** qualifier. The presence of the **`_Atomic`** qualifier designates an atomic type. The size, representation, and alignment of an atomic type need not be the same as those of the corresponding unqualified type. Therefore, this Standard explicitly uses the phrase ‘‘atomic, qualified or unqualified type’’ whenever the atomic version of a type is permitted along with the other qualified versions of a type. The phrase ‘‘qualified or unqualified type’’, without specific mention of atomic, does not include the atomic types.

28 A pointer to **`void`** shall have the same representation and alignment requirements as a pointer to a character type.[48] Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types shall have the same representation and alignment requirements as each other. All pointers to union types shall have the same representation and alignment requirements as each other. Pointers to other types need not have the same representation or alignment requirements.

29 EXAMPLE 1 The type designated as ‘‘**`float *`**’’ has type ‘‘pointer to **`float`**’’. Its type category is pointer, not a floating type. The const-qualified version of this type is designated as ‘‘**`float * const`**’’ whereas the type designated as ‘‘**`const float *`**’’ is not a qualified type — its type is ‘‘pointer to const-

47) See 6.7.3 regarding qualified array and function types.

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

qualified `float`'' and is a pointer to a qualified type.

30 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.[49)]

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.[50)] 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.[51)] The value of a structure or union object is never a trap

49) 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 $2^{\mathtt{CHAR\_BIT}} - 1$.

50) 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.

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.[52] 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 are done with **`memory_order_seq_cst`** semantics.

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.[53]

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

51) Thus, for example, structure assignment need not copy any padding bits.

52) 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`** may distinguish between them.

53) 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.

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.[54] 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.

54) 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.

6.2.7 Compatible type and composite type

1 Two types have *compatible type* if their types are the same. Additional rules for determining whether two types are compatible are described in 6.7.2 for type specifiers, in 6.7.3 for type qualifiers, and in 6.7.6 for declarators.[55] Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if their tags and members satisfy the following requirements: If one is declared with a tag, the other shall be declared with the same tag. If both are completed anywhere within their respective translation units, then the following additional requirements apply: there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types; if one member of the pair is declared with an alignment specifier, the other is declared with an equivalent alignment specifier; and if one member of the pair is declared with a name, the other is declared with the same name. For two structures, corresponding members shall be declared in the same order. For two structures or unions, corresponding bit-fields shall have the same widths. For two enumerations, corresponding members shall have the same values.

2 All declarations that refer to the same object or function shall have compatible type; otherwise, the behavior is undefined.

3 A *composite type* can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:

— If both types are array types, the following rules are applied:

  - If one type is an array of known constant size, the composite type is an array of that size.
  - Otherwise, if one type is a variable length array whose size is specified by an expression that is not evaluated, the behavior is undefined.
  - Otherwise, if one type is a variable length array whose size is specified, the composite type is a variable length array of that size.
  - Otherwise, if one type is a variable length array of unspecified size, the composite type is a variable length array of unspecified size.
  - Otherwise, both types are arrays of unknown size and the composite type is an array of unknown size.

  The element type of the composite type is the composite type of the two element types.

— If only one type is a function type with a parameter type list (a function prototype), the composite type is a function prototype with the parameter type list.

55) Two types need not be identical to be compatible.

— If both types are function types with parameter type lists, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.

These rules apply recursively to the types from which the two types are derived.

4 For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible,[56] if the prior declaration specifies internal or external linkage, the type of the identifier at the later declaration becomes the composite type.

Forward references: array declarators (6.7.6.2).

5 EXAMPLE Given the following two file scope declarations:

```
int f(int (*)(), double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

The resulting composite type for the function is:

```
int f(int (*)(char *), double (*)[3]);
```

6.2.8 Alignment of objects

1 Complete object types have alignment requirements which place restrictions on the addresses at which objects of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type: stricter alignment can be requested using the **`_Alignas`** keyword.

2 A *fundamental alignment* is represented by an alignment less than or equal to the greatest alignment supported by the implementation in all contexts, which is equal to **`_Alignof (max_align_t)`**.

3 An *extended alignment* is represented by an alignment greater than **`_Alignof (max_align_t)`**. It is implementation-defined whether any extended alignments are supported and the contexts in which they are supported. A type having an extended alignment requirement is an *over-aligned* type.[57]

4 Alignments are represented as values of the type **`size_t`**. Valid alignments include only those values returned by an **`_Alignof`** expression for fundamental types, plus an additional implementation-defined set of values, which may be empty. Every valid alignment value shall be a nonnegative integral power of two.

56) As specified in 6.2.1, the later declaration might hide the prior declaration.

57) Every over-aligned type is, or contains, a structure or union type with a member to which an extended alignment has been applied.

5 Alignments have an order from *weaker* to *stronger* or *stricter* alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any weaker valid alignment requirement.

6 The alignment requirement of a complete type can be queried using an **`_Alignof`** expression. The types **`char`**, **`signed char`**, and **`unsigned char`** shall have the weakest alignment requirement.

7 Comparing alignments is meaningful and provides the obvious results:

— Two alignments are equal when their numeric values are equal.

— Two alignments are different when their numeric values are not equal.

— When an alignment is larger than another it represents a stricter alignment.

6.3 Conversions

1 Several operators convert operand values from one type to another automatically. This subclause specifies the result required from such an *implicit conversion*, as well as those that result from a cast operation (an *explicit conversion*). The list in 6.3.1.8 summarizes the conversions performed by most ordinary operators; it is supplemented as required by the discussion of each operator in 6.5.

2 Conversion of an operand value to a compatible type causes no change to the value or the representation.

Forward references: cast operators (6.5.4).

6.3.1 Arithmetic operands

6.3.1.1 Boolean, characters, and integers

1 Every integer type has an *integer conversion rank* defined as follows:

— No two signed integer types shall have the same rank, even if they have the same representation.

— The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.

— 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*.[58] All other types are unchanged by the integer promotions.

3 The integer promotions preserve value including sign. As discussed earlier, whether a ‘‘plain’’ **char** is treated as signed 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 if the value compares equal to 0; otherwise, the result is 1.[59]

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.[60]

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.

6.3.1.4 Real floating and integer

1 When a finite value of real floating type is converted to an integer type other than **_Bool**, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the behavior is undefined.[61]

---

58) 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.

59) NaNs do not compare equal to 0 and thus convert to 1.

60) The rules describe arithmetic on the mathematical value, not the value of a given type of expression.

61) The remaindering operation performed when a value of integer type is converted to unsigned type need not be performed when a value of real floating type is converted to unsigned type. Thus, the range of portable real floating values is (−1, **U***type***_MAX**+1).

2 When a value of integer type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is undefined. Results of some implicit conversions may be represented in greater range and precision than that required by the new type (see 6.3.1.8 and 6.8.6.4).

6.3.1.5 Real floating types

1 When a value of real floating type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is undefined. Results of some implicit conversions may be represented in greater range and precision than that required by the new type (see 6.3.1.8 and 6.8.6.4).

6.3.1.6 Complex types

1 When a value of complex type is converted to another complex type, both the real and imaginary parts follow the conversion rules for the corresponding real types.

6.3.1.7 Real and complex

1 When a value of real type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion to the corresponding real type and the imaginary part of the complex result value is a positive zero or an unsigned zero.

2 When a value of complex type is converted to a real type, the imaginary part of the complex value is discarded and the value of the real part is converted according to the conversion rules for the corresponding real type.

6.3.1.8 Usual arithmetic conversions

1 Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a *common real type* for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. This pattern is called the *usual arithmetic conversions*:

> First, if the corresponding real type of either operand is **`long double`**, the other operand is converted, without change of type domain, to a type whose

corresponding real type is **`long double`**.

Otherwise, if the corresponding real type of either operand is **`double`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`double`**.

Otherwise, if the corresponding real type of either operand is **`float`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`float`**.[62)]

Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:

If both operands have the same type, then no further conversion is needed.

Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.

Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

2 The values of floating operands and of the results of floating expressions may be represented in greater range and precision than that required by the type; the types are not changed thereby.[63)]

62) For example, addition of a **`double _Complex`** and a **`float`** entails just the conversion of the **`float`** operand to **`double`** (and yields a **`double _Complex`** result).

63) The cast and assignment operators are still required to remove extra range and precision.

6.3.2 Other operands

6.3.2.1 Lvalues, arrays, and function designators

1 An *lvalue* is an expression (with an object type other than **void**) that potentially designates an object;[64] if an lvalue does not designate an object when it is evaluated, the behavior is undefined. When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.

2 Except when it is the operand of the **sizeof** operator, the **_Alignof** operator, the unary **&** operator, the **++** operator, the **--** operator, or the left operand of the **.** operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called *lvalue conversion*. If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; additionally, if the lvalue has atomic type, the value has the non-atomic version of the type of the lvalue; otherwise, the value has the type of the lvalue. If the lvalue has an incomplete type and does not have array type, the behavior is undefined. If the lvalue designates an object of automatic storage duration that could have been declared with the **register** storage class (never had its address taken), and that object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined.

3 Except when it is the operand of the **sizeof** operator, the **_Alignof** operator, or the unary **&** operator, or is a string literal used to initialize an array, an expression that has type ‘‘array of *type*’’ is converted to an expression with type ‘‘pointer to *type*’’ that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.

4 A *function designator* is an expression that has function type. Except when it is the operand of the **sizeof** operator, the **_Alignof** operator,[65] or the unary **&** operator, a function designator with type ‘‘function returning *type*’’ is converted to an expression that

---

64) The name ‘‘lvalue’’ comes originally from the assignment expression **E1 = E2**, in which the left operand **E1** is required to be a (modifiable) lvalue. It is perhaps better considered as representing an object ‘‘locator value’’. What is sometimes called ‘‘rvalue’’ is in this International Standard described as the ‘‘value of an expression’’.

An obvious example of an lvalue is an identifier of an object. As a further example, if **E** is a unary expression that is a pointer to an object, ***E** is an lvalue that designates the object to which **E** points.

65) Because this conversion does not occur, the operand of the **sizeof** or **_Alignof** operator remains a function designator and violates the constraints in 6.5.3.4.

has type ‘‘pointer to function returning *type*’’.

Forward references: address and indirection operators (6.5.3.2), assignment operators (6.5.16), common definitions **<stddef.h>** (7.19), initialization (6.7.9), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), the **sizeof** and **_Alignof** operators (6.5.3.4), structure and union members (6.5.2.3).

6.3.2.2 void

1 The (nonexistent) value of a *void expression* (an expression that has type **void**) shall not be used in any way, and implicit or explicit conversions (except to **void**) shall not be applied to such an expression. If an expression of any other type is evaluated as a void expression, its value or designator is discarded. (A void expression is evaluated for its side effects.)

6.3.2.3 Pointers

1 A pointer to **void** may be converted to or from a pointer to any object type. A pointer to any object type may be converted to a pointer to **void** and back again; the result shall compare equal to the original pointer.

2 For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q*-qualified version of the type; the values stored in the original and converted pointers shall compare equal.

3 An integer constant expression with the value 0, or such an expression cast to type **void ***, is called a *null pointer constant*.[66] If a null pointer constant is converted to a pointer type, the resulting pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.

4 Conversion of a null pointer to another pointer type yields a null pointer of that type. Any two null pointers shall compare equal.

5 An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.[67]

6 Any pointer type may be converted to an integer type. Except as previously specified, the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type.

---

66) The macro **NULL** is defined in **<stddef.h>** (and other headers) as a null pointer constant; see 7.19.

67) The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment.

7 A pointer to an object type may be converted to a pointer to a different object type. If the resulting pointer is not correctly aligned[68] for the referenced type, the behavior is undefined. Otherwise, when converted back again, the result shall compare equal to the original pointer. When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object.

8 A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the referenced type, the behavior is undefined.

Forward references: cast operators (6.5.4), equality operators (6.5.9), integer types capable of holding object pointers (7.20.1.4), simple assignment (6.5.16.1).

68) In general, the concept ''correctly aligned'' is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which in turn is correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C.

6.4 Lexical elements

Syntax

1 *token:*
  keyword
  identifier
  constant
  string-literal
  punctuator

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

Constraints

2 Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a constant, a string literal, or a punctuator.

Semantics

3 A *token* is the minimal lexical element of the language in translation phases 7 and 8. The categories of tokens are: keywords, identifiers, constants, string literals, and punctuators. A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing tokens are: header names, identifiers, preprocessing numbers, character constants, string literals, punctuators, and single non-white-space characters that do not lexically match the other preprocessing token categories.[69)] If a `'` or a `"` character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by *white space*; this consists of comments (described later), or *white-space characters* (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in 6.10, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space may appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

---

69) An additional category, placemarkers, is used internally in translation phase 4 (see 6.10.3.3); it cannot occur in source files.

4 If the input stream has been parsed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token. There is one exception to this rule: header name preprocessing tokens are recognized only within **#include** preprocessing directives and in implementation-defined locations within **#pragma** directives. In such contexts, a sequence of characters that could be either a header name or a string literal is recognized as the former.

5 EXAMPLE 1 The program fragment **1Ex** is parsed as a preprocessing number token (one that is not a valid floating or integer constant token), even though a parse as the pair of preprocessing tokens **1** and **Ex** might produce a valid expression (for example, if **Ex** were a macro defined as **+1**). Similarly, the program fragment **1E1** is parsed as a preprocessing number (one that is a valid floating constant token), whether or not **E** is a macro name.

6 EXAMPLE 2 The program fragment **x+++++y** is parsed as **x ++ ++ + y**, which violates a constraint on increment operators, even though the parse **x ++ + ++ y** might yield a correct expression.

Forward references: character constants (6.4.4.4), comments (6.4.9), expressions (6.5), floating constants (6.4.4.2), header names (6.4.7), macro replacement (6.10.3), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), preprocessing directives (6.10), preprocessing numbers (6.4.8), string literals (6.4.5).

6.4.1 Keywords

Syntax

1 *keyword:* one of

```
auto        * if          unsigned
break         inline      void
case          int         volatile
char          long        while
const         register    _Alignas
continue      restrict    _Alignof
default       return      _Atomic
do            short       _Bool
double        signed      _Complex
else          sizeof      _Generic
enum          static      _Imaginary
extern        struct      _Noreturn
float         switch      _Static_assert
for           typedef     _Thread_local
goto          union
```

Semantics

2 The above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords, and shall not be used otherwise. The keyword **_Imaginary** is reserved for

specifying imaginary types.[70)]

6.4.2 Identifiers

6.4.2.1 General

Syntax

1 *identifier:*
    identifier-nondigit
    identifier identifier-nondigit
    identifier digit

identifier-nondigit:
    nondigit
    universal-character-name
    other implementation-defined characters

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
```

digit: one of

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

Semantics

2 An identifier is a sequence of nondigit characters (including the underscore `_`, the lowercase and uppercase Latin letters, and other characters) and digits, which designates one or more entities as described in 6.2.1. Lowercase and uppercase letters are distinct. There is no specific limit on the maximum length of an identifier.

3 Each universal character name in an identifier shall designate a character whose encoding in ISO/IEC 10646 falls into one of the ranges specified in D.1.[71)] The initial character shall not be a universal character name designating a character whose encoding falls into one of the ranges specified in D.2. An implementation may allow multibyte characters that are not part of the basic source character set to appear in identifiers; which characters and their correspondence to universal character names is implementation-defined.

---

70) One possible specification for imaginary types appears in annex G.

71) On systems in which linkers cannot accept extended characters, an encoding of the universal character name may be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters may be used to encode the `\u` in a universal character name. Extended characters may produce a long external identifier.

4 When preprocessing tokens are converted to tokens during translation phase 7, if a preprocessing token could be converted to either a keyword or an identifier, it is converted to a keyword.

Implementation limits

5 As discussed in 5.2.4.1, an implementation may limit the number of significant initial characters in an identifier; the limit for an *external name* (an identifier that has external linkage) may be more restrictive than that for an *internal name* (a macro name or an identifier that does not have external linkage). The number of significant characters in an identifier is implementation-defined.

6 Any identifiers that differ in a significant character are different identifiers. If two identifiers differ only in nonsignificant characters, the behavior is undefined.

Forward references: universal character names (6.4.3), macro replacement (6.10.3).

6.4.2.2 Predefined identifiers

Semantics

1 The identifier **`__func__`** shall be implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration

```
static const char __func__[] = "function-name";
```

appeared, where *function-name* is the name of the lexically-enclosing function.[72)]

2 This name is encoded as if the implicit declaration had been written in the source character set and then translated into the execution character set as indicated in translation phase 5.

3 EXAMPLE Consider the code fragment:

```
#include <stdio.h>
void myfunc(void)
{
      printf("%s\n", __func__);
      /* ... */
}
```

Each time the function is called, it will print to the standard output stream:

```
myfunc
```

Forward references: function definitions (6.9.1).

72) Since the name **`__func__`** is reserved for any use by the implementation (7.1.3), if any other identifier is explicitly declared using the name **`__func__`**, the behavior is undefined.

6.4.3 Universal character names

Syntax

1 *universal-character-name:*
  `\u` *hex-quad*
  `\U` *hex-quad hex-quad*

hex-quad:
  hexadecimal-digit hexadecimal-digit
    hexadecimal-digit hexadecimal-digit

Constraints

2 A universal character name shall not specify a character whose short identifier is less than 00A0 other than 0024 (`$`), 0040 (`@`), or 0060 (`` ` ``), nor one in the range D800 through DFFF inclusive.[73)]

Description

3 Universal character names may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set.

Semantics

4 The universal character name `\U`*nnnnnnnn* designates the character whose eight-digit short identifier (as specified by ISO/IEC 10646) is *nnnnnnnn*.[74)] Similarly, the universal character name `\u`*nnnn* designates the character whose four-digit short identifier is *nnnn* (and whose eight-digit short identifier is 0000*nnnn*).

---

73) The disallowed characters are the characters in the basic character set and the code positions reserved by ISO/IEC 10646 for control characters, the character DELETE, and the S-zone (reserved for use by UTF−16).

74) Short identifiers for characters were first specified in ISO/IEC 10646−1/AMD9:1997.

6.4.4 Constants

Syntax

1 *constant:*
    integer-constant
    floating-constant
    enumeration-constant
    character-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 $integer\text{-}suffix_{opt}$
    octal-constant $integer\text{-}suffix_{opt}$
    hexadecimal-constant $integer\text{-}suffix_{opt}$

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$_{opt}$
unsigned-suffix long-long-suffix
long-suffix unsigned-suffix$_{opt}$
long-long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
`u U`

long-suffix: one of
`l L`

long-long-suffix: one of
`ll LL`

Description

2 An integer constant begins with a digit, but has no period or exponent part. It may have a prefix that specifies its base and a suffix that specifies its type.

3 A decimal constant begins with a nonzero digit and consists of a sequence of decimal digits. An octal constant consists of the prefix **`0`** optionally followed by a sequence of the digits **`0`** through **`7`** only. A hexadecimal constant consists of the prefix **`0x`** or **`0X`** followed by a sequence of the decimal digits and the letters **`a`** (or **`A`**) through **`f`** (or **`F`**) with values 10 through 15 respectively.

Semantics

4 The value of a decimal constant is computed base 10; that of an octal constant, base 8; that of a hexadecimal constant, base 16. The lexically first digit is the most significant.

5 The type of an integer constant is the first of the corresponding list in which its value can be represented.

Suffix	Decimal Constant	Octal or Hexadecimal Constant

none	`int`
	`long int`
	`long long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` or `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` or `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

Both `u` or `U` and `l` or `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` or `LL`	`long long int`	`long long int` `unsigned long long int`
Both `u` or `U` and `ll` or `LL`	`unsigned long long int`	`unsigned long long int`

6 If an integer constant cannot be represented by any type in its list, it may have an extended integer type, if the extended integer type can represent its value. If all of the types in the list for the constant are signed, the extended integer type shall be signed. If all of the types in the list for the constant are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. If an integer constant cannot be represented by any type in its list and has no extended integer type, then the integer constant has no type.

6.4.4.2 Floating constants

Syntax

1 *floating-constant:*
decimal-floating-constant
hexadecimal-floating-constant

decimal-floating-constant:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$

hexadecimal-floating-constant:
hexadecimal-prefix hexadecimal-fractional-constant
binary-exponent-part floating-suffix$_{opt}$
hexadecimal-prefix hexadecimal-digit-sequence
binary-exponent-part floating-suffix$_{opt}$

fractional-constant:
digit-sequence$_{opt}$ **.** *digit-sequence*
digit-sequence **.**

exponent-part:
e *sign*$_{opt}$ *digit-sequence*
E *sign*$_{opt}$ *digit-sequence*

sign: one of
+ -

digit-sequence:
digit
digit-sequence digit

hexadecimal-fractional-constant:
hexadecimal-digit-sequence$_{opt}$ **.**
hexadecimal-digit-sequence
hexadecimal-digit-sequence **.**

binary-exponent-part:
p *sign*$_{opt}$ *digit-sequence*
P *sign*$_{opt}$ *digit-sequence*

hexadecimal-digit-sequence:
hexadecimal-digit
hexadecimal-digit-sequence hexadecimal-digit

floating-suffix: one of
f l F L

Description

2 A floating constant has a *significand part* that may be followed by an *exponent part* and a suffix that specifies its type. The components of the significand part may include a digit sequence representing the whole-number part, followed by a period (**.**), followed by a digit sequence representing the fraction part. The components of the exponent part are an **e**, **E**, **p**, or **P** followed by an exponent consisting of an optionally signed digit sequence. Either the whole-number part or the fraction part has to be present; for decimal floating constants, either the period or the exponent part has to be present.

Semantics

3 The significand part is interpreted as a (decimal or hexadecimal) rational number; the digit sequence in the exponent part is interpreted as a decimal integer. For decimal floating constants, the exponent indicates the power of 10 by which the significand part is to be scaled. For hexadecimal floating constants, the exponent indicates the power of 2 by which the significand part is to be scaled. For decimal floating constants, and also for hexadecimal floating constants when **FLT_RADIX** is not a power of 2, the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner. For hexadecimal floating constants when **FLT_RADIX** is a power of 2, the result is correctly rounded.

4 An unsuffixed floating constant has type **double**. If suffixed by the letter **f** or **F**, it has type **float**. If suffixed by the letter **l** or **L**, it has type **long double**.

5 Floating constants are converted to internal format as if at translation-time. The conversion of a floating constant shall not raise an exceptional condition or a floating-point exception at execution time. All floating constants of the same source form[75] shall convert to the same internal format with the same value.

Recommended practice

6 The implementation should produce a diagnostic message if a hexadecimal constant cannot be represented exactly in its evaluation format; the implementation should then proceed with the translation of the program.

7 The translation-time conversion of floating constants should match the execution-time conversion of character strings by library functions, such as **strtod**, given matching inputs suitable for both conversions, the same result format, and default execution-time rounding.[76]

---

75) **1.23**, **1.230**, **123e-2**, **123e-02**, and **1.23L** are all different source forms and thus need not convert to the same internal format and value.

76) The specification for the library functions recommends more accurate conversion than required for floating constants (see 7.22.1.3).

6.4.4.3 Enumeration constants

Syntax

1 *enumeration-constant:*
  identifier

Semantics

2 An identifier declared as an enumeration constant has type **`int`**.

Forward references: enumeration specifiers (6.7.2.2).

6.4.4.4 Character constants

Syntax

1 *character-constant:*
  `'` *c-char-sequence* **`'`**
  `L'` *c-char-sequence* **`'`**
  `u'` *c-char-sequence* **`'`**
  `U'` *c-char-sequence* **`'`**

c-char-sequence:
  c-char
  c-char-sequence c-char

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

escape-sequence:
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence
  universal-character-name

simple-escape-sequence: one of
  `\'  \"  \?  \\`
  `\a  \b  \f  \n  \r  \t  \v`

octal-escape-sequence:
  `\` *octal-digit*
  `\` *octal-digit octal-digit*
  `\` *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
\x *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

Description

2 An integer character constant is a sequence of one or more multibyte characters enclosed in single-quotes, as in **'x'**. A wide character constant is the same, except prefixed by the letter **L**, **u**, or **U**. With a few exceptions detailed later, the elements of the sequence are any members of the source character set; they are mapped in an implementation-defined manner to members of the execution character set.

3 The single-quote **'**, the double-quote **"**, the question-mark **?**, the backslash ****, and arbitrary integer values are representable according to the following table of escape sequences:

single quote **'**	**\'**
double quote **"**	**\"**
question mark **?**	**\?**
backslash ****	****
octal character	*****octal digits*
hexadecimal character	**\x** *hexadecimal digits*

4 The double-quote **"** and question-mark **?** are representable either by themselves or by the escape sequences **\"** and **\?**, respectively, but the single-quote **'** and the backslash **** shall be represented, respectively, by the escape sequences **\'** and ****.

5 The octal digits that follow the backslash in an octal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the octal integer so formed specifies the value of the desired character or wide character.

6 The hexadecimal digits that follow the backslash and the letter **x** in a hexadecimal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the hexadecimal integer so formed specifies the value of the desired character or wide character.

7 Each octal or hexadecimal escape sequence is the longest sequence of characters that can constitute the escape sequence.

8 In addition, characters not in the basic character set are representable by universal character names and certain nongraphic characters are representable by escape sequences consisting of the backslash **** followed by a lowercase letter: **\a**, **\b**, **\f**, **\n**, **\r**, **\t**, and **\v**.[77)]

Constraints

9 The value of an octal or hexadecimal escape sequence shall be in the range of representable values for the corresponding type:

Prefix	Corresponding Type
none	**unsigned char**
L	the unsigned type corresponding to **wchar_t**
u	**char16_t**
U	**char32_t**

Semantics

10 An integer character constant has type **int**. The value of an integer character constant containing a single character that maps to a single-byte execution character is the numerical value of the representation of the mapped character interpreted as an integer. The value of an integer character constant containing more than one character (e.g., **'ab'**), or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined. If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type **char** whose value is that of the single character or escape sequence is converted to type **int**.

11 A wide character constant prefixed by the letter **L** has type **wchar_t**, an integer type defined in the **<stddef.h>** header; a wide character constant prefixed by the letter **u** or **U** has type **char16_t** or **char32_t**, respectively, unsigned integer types defined in the **<uchar.h>** header. The value of a wide character constant containing a single multibyte character that maps to a single member of the extended execution character set is the wide character corresponding to that multibyte character, as defined by the **mbtowc**, **mbrtoc16**, or **mbrtoc32** function as appropriate for its type, with an implementation-defined current locale. The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined.

12 EXAMPLE 1 The construction **'\0'** is commonly used to represent the null character.

13 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**.

77) The semantics of these characters were discussed in 5.2.2. If any other character follows a backslash, the result is not a token and a diagnostic is required. See ‘‘future language directions’’ (6.11.4).

14 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'`** may 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.)

15 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).

6.4.5 String literals

Syntax

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

encoding-prefix:
    `u8`
    `u`
    `U`
    `L`

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.

Description

3 A *character string literal* is a sequence of zero or more multibyte characters enclosed in double-quotes, as in **`"xyz"`**. A *UTF−8 string literal* is the same, except prefixed by **`u8`**. A *wide string literal* is the same, except prefixed by the letter **`L`**, **`u`**, or **`U`**.

4 The same considerations apply to each element of the sequence in a string literal as if it were in an integer character constant (for a character or UTF−8 string literal) or a wide character constant (for a wide string literal), except that the single-quote **`'`** is representable either by itself or by the escape sequence **`\'`**, but the double-quote **`"`** shall

be represented by the escape sequence `\"`.

Semantics

5 In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character and identically-prefixed string literal tokens are concatenated into a single multibyte character sequence. If any of the tokens has an encoding prefix, the resulting multibyte character sequence is treated as having the same prefix; otherwise, it is treated as a character string literal. Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence are implementation-defined.

6 In translation phase 7, a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals.[78)] The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. For character string literals, the array elements have type **`char`**, and are initialized with the individual bytes of the multibyte character sequence. For UTF−8 string literals, the array elements have type **`char`**, and are initialized with the characters of the multibyte character sequence, as encoded in UTF−8. For wide string literals prefixed by the letter **`L`**, the array elements have type **`wchar_t`** and are initialized with the sequence of wide characters corresponding to the multibyte character sequence, as defined by the **`mbstowcs`** function with an implementation-defined current locale. For wide string literals prefixed by the letter **`u`** or **`U`**, the array elements have type **`char16_t`** or **`char32_t`**, respectively, and are initialized with the sequence of wide characters corresponding to the multibyte character sequence, as defined by successive calls to the **`mbrtoc16`**, or **`mbrtoc32`** function as appropriate for its type, with an implementation-defined current locale. The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined.

7 It is unspecified whether these arrays are distinct provided their elements have the appropriate values. If the program attempts to modify such an array, the behavior is undefined.

8 EXAMPLE 1 This pair of adjacent character string literals

```
"\x12" "3"
```

produces a single character string literal containing the two characters whose values are `'\x12'` and `'3'`, because escape sequences are converted into single members of the execution character set just prior to adjacent string literal concatenation.

9 EXAMPLE 2 Each of the sequences of adjacent string literal tokens

78) A string literal need not be a string (see 7.1.1), because a null character may be embedded in it by a `\0` escape sequence.

```
"a" "b" L"c"
"a" L"b" "c"
L"a" "b" L"c"
L"a" L"b" L"c"
```

is equivalent to the string literal

```
L"abc"
```

Likewise, each of the sequences

```
"a" "b" u"c"
"a" u"b" "c"
u"a" "b" u"c"
u"a" u"b" u"c"
```

is equivalent to

```
u"abc"
```

Forward references: common definitions **<stddef.h>** (7.19), the **mbstowcs** function (7.22.8.1), Unicode utilities **<uchar.h>** (7.28).

6.4.6 Punctuators

Syntax

1 *punctuator:* one of

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

Semantics

2 A punctuator is a symbol that has independent syntactic and semantic significance. Depending on context, it may specify an operation to be performed (which in turn may yield a value or a function designator, produce a side effect, or some combination thereof) in which case it is known as an *operator* (other forms of operator also exist in some contexts). An *operand* is an entity on which an operator acts.

3 In all aspects of the language, the six tokens[79)]

```
<:  :>  <%  %>  %:  %:%:
```

behave, respectively, the same as the six tokens

```
[  ]  {  }  #  ##
```

except for their spelling.[80)]

Forward references: expressions (6.5), declarations (6.7), preprocessing directives (6.10), statements (6.8).

6.4.7 Header names

Syntax

1 *header-name:*
  < *h-char-sequence* **>**
  " *q-char-sequence* **"**

h-char-sequence:
  h-char
  h-char-sequence h-char

h-char:
  any member of the source character set except
    the new-line character and **>**

q-char-sequence:
  q-char
  q-char-sequence q-char

q-char:
  any member of the source character set except
    the new-line character and **"**

Semantics

2 The sequences in both forms of header names are mapped in an implementation-defined manner to headers or external source file names as specified in 6.10.2.

3 If the characters `'`, `\`, `"`, `//`, or `/*` occur in the sequence between the `<` and `>` delimiters, the behavior is undefined. Similarly, if the characters `'`, `\`, `//`, or `/*` occur in the

---

79) These tokens are sometimes called ‘‘digraphs’’.

80) Thus `[` and `<:` behave differently when ‘‘stringized’’ (see 6.10.3.2), but can otherwise be freely interchanged.

sequence between the `"` delimiters, the behavior is undefined.[81)] Header name preprocessing tokens are recognized only within **`#include`** preprocessing directives and in implementation-defined locations within **`#pragma`** directives.[82)]

4 EXAMPLE The following sequence of characters:

```
0x3<1/a.h>1e2
#include <1/a.h>
#define const.member@$
```

forms the following sequence of preprocessing tokens (with each individual preprocessing token delimited by a *{* on the left and a *}* on the right).

```
{0x3}{<}{1}{/}{a}{.}{h}{>}{1e2}
{#}{include} {<1/a.h>}
{#}{define} {const}{.}{member}{@}{$}
```

Forward references: source file inclusion (6.10.2).

6.4.8 Preprocessing numbers

Syntax

1 *pp-number:*
 digit
 . *digit*
 pp-number digit
 pp-number identifier-nondigit
 pp-number **e** *sign*
 pp-number **E** *sign*
 pp-number **p** *sign*
 pp-number **P** *sign*
 pp-number **.**

Description

2 A preprocessing number begins with a digit optionally preceded by a period (**.**) and may be followed by valid identifier characters and the character sequences **e+**, **e-**, **E+**, **E-**, **p+**, **p-**, **P+**, or **P-**.

3 Preprocessing number tokens lexically include all floating and integer constant tokens.

Semantics

4 A preprocessing number does not have type or a value; it acquires both after a successful conversion (as part of translation phase 7) to a floating constant token or an integer constant token.

---

81) Thus, sequences of characters that resemble escape sequences cause undefined behavior.

82) For an example of a header name preprocessing token used in a **`#pragma`** directive, see 6.10.9.

6.4.9 Comments

1 Except within a character constant, a string literal, or a comment, the characters **/*** introduce a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters ***/** that terminate it.[83)]

2 Except within a character constant, a string literal, or a comment, the characters **//** introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

3 EXAMPLE

```
"a//b"                  // four-character string literal
#include "//e"          // undefined behavior
// */                   // comment, not syntax error
f = g/**//h;            // equivalent to f = g / h;
//\
i();                    // part of a two-line comment
/\
/ j();                  // part of a two-line comment
#define glue(x,y) x##y
glue(/,/) k();          // syntax error, not comment
/*//*/ l();             // equivalent to l();
m = n//**/o
  + p;                  // equivalent to m = n + p;
```

83) Thus, **/*** ... ***/** comments do not nest.

6.5 Expressions

1 An *expression* is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof. The value computations of the operands of an operator are sequenced before the value computation of the result of the operator.

2 If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.[84)]

3 The grouping of operators and operands is indicated by the syntax.[85)] Except as specified later, side effects and value computations of subexpressions are unsequenced.[86)]

4 Some operators (the unary operator `~`, and the binary operators `<<`, `>>`, `&`, `^`, and `|`, collectively described as *bitwise operators*) are required to have operands that have integer type. These operators yield values that depend on the internal representations of integers, and have implementation-defined and undefined aspects for signed types.

5 If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

---

84) This paragraph renders undefined statement expressions such as

```
i = ++i + 1;
a[i++] = i;
```

while allowing

```
i = i + 1;
a[i] = i;
```

85) The syntax specifies the precedence of operators in the evaluation of an expression, which is the same as the order of the major subclauses of this subclause, highest precedence first. Thus, for example, the expressions allowed as the operands of the binary `+` operator (6.5.6) are those expressions defined in 6.5.1 through 6.5.6. The exceptions are cast expressions (6.5.4) as operands of unary operators (6.5.3), and an operand contained between any of the following pairs of operators: grouping parentheses `()` (6.5.1), subscripting brackets `[]` (6.5.2.1), function-call parentheses `()` (6.5.2.2), and the conditional operator `? :` (6.5.15).

Within each major subclause, the operators have the same precedence. Left- or right-associativity is indicated in each subclause by the syntax for the expressions discussed therein.

86) In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations.

6 The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[87] If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using **`memcpy`** or **`memmove`**, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[88]

— a type compatible with the effective type of the object,

— a qualified version of a type compatible with the effective type of the object,

— a type that is the signed or unsigned type corresponding to the effective type of the object,

— a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,

— an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or

— a character type.

8 A floating expression may be *contracted*, that is, evaluated as though it were a single operation, thereby omitting rounding errors implied by the source code and the expression evaluation method.[89] The **`FP_CONTRACT`** pragma in **`<math.h>`** provides a way to disallow contracted expressions. Otherwise, whether and how expressions are contracted is implementation-defined.[90]

Forward references: the **`FP_CONTRACT`** pragma (7.12.2), copying functions (7.24.2).

---

87) Allocated objects have no declared type.

88) The intent of this list is to specify those circumstances in which an object may or may not be aliased.

89) The intermediate operations in the contracted expression are evaluated as if to infinite range and precision, while the final operation is rounded to the format determined by the expression evaluation method. A contracted expression might also omit the raising of floating-point exceptions.

90) This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple C operators. As contractions potentially undermine predictability, and can even decrease accuracy for containing expressions, their use needs to be well-defined and clearly documented.

6.5.1 Primary expressions

Syntax

1 *primary-expression:*
identifier
constant
string-literal
`(` *expression* `)`
generic-selection

Semantics

2 An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator).[91)]

3 A constant is a primary expression. Its type depends on its form and value, as detailed in 6.4.4.

4 A string literal is a primary expression. It is an lvalue with type as detailed in 6.4.5.

5 A parenthesized expression is a primary expression. Its type and value are identical to those of the unparenthesized expression. It is an lvalue, a function designator, or a void expression if the unparenthesized expression is, respectively, an lvalue, a function designator, or a void expression.

6 A generic selection is a primary expression. Its type and value depend on the selected generic association, as detailed in the following subclause.

Forward references: declarations (6.7).

6.5.1.1 Generic selection

Syntax

1 *generic-selection:*
`_Generic` `(` *assignment-expression* `,` *generic-assoc-list* `)`

generic-assoc-list:
generic-association
generic-assoc-list `,` *generic-association*

generic-association:
type-name `:` *assignment-expression*
`default` `:` *assignment-expression*

91) Thus, an undeclared identifier is a violation of the syntax.

Constraints

2 A generic selection shall have no more than one **`default`** generic association. The type name in a generic association shall specify a complete object type other than a variably modified type. No two generic associations in the same generic selection shall specify compatible types. The controlling expression of a generic selection shall have type compatible with at most one of the types named in its generic association list. If a generic selection has no **`default`** generic association, its controlling expression shall have type compatible with exactly one of the types named in its generic association list.

Semantics

3 The controlling expression of a generic selection is not evaluated. If a generic selection has a generic association with a type name that is compatible with the type of the controlling expression, then the result expression of the generic selection is the expression in that generic association. Otherwise, the result expression of the generic selection is the expression in the **`default`** generic association. None of the expressions from any other generic association of the generic selection is evaluated.

4 The type and value of a generic selection are identical to those of its result expression. It is an lvalue, a function designator, or a void expression if its result expression is, respectively, an lvalue, a function designator, or a void expression.

5 EXAMPLE The **`cbrt`** type-generic macro could be implemented as follows:

```
#define cbrt(X) _Generic((X),                   \
                        long double: cbrtl,     \
                        default: cbrt,          \
                        float: cbrtf            \
                        )(X)
```

6.5.2 Postfix operators

Syntax

1 *postfix-expression:*
primary-expression
postfix-expression **[** *expression* **]**
postfix-expression **(** *argument-expression-list$_{opt}$* **)**
postfix-expression **.** *identifier*
postfix-expression **->** *identifier*
postfix-expression **++**
postfix-expression **--**
(*type-name* **)** **{** *initializer-list* **}**
(*type-name* **)** **{** *initializer-list* **, }**

argument-expression-list:
  assignment-expression
  argument-expression-list **,** *assignment-expression*

6.5.2.1 Array subscripting

Constraints

1 One of the expressions shall have type ''pointer to complete object *type*'', the other expression shall have integer type, and the result has type ''*type*''.

Semantics

2 A postfix expression followed by an expression in square brackets **[]** is a subscripted designation of an element of an array object. The definition of the subscript operator **[]** is that **E1[E2]** is identical to **(*((E1)+(E2)))**. Because of the conversion rules that apply to the binary **+** operator, if **E1** is an array object (equivalently, a pointer to the initial element of an array object) and **E2** is an integer, **E1[E2]** designates the **E2**-th element of **E1** (counting from zero).

3 Successive subscript operators designate an element of a multidimensional array object. If **E** is an *n*-dimensional array ($n \geq 2$) with dimensions $i \times j \times \cdots \times k$, then **E** (used as other than an lvalue) is converted to a pointer to an $(n-1)$-dimensional array with dimensions $j \times \cdots \times k$. If the unary ***** operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the referenced $(n-1)$-dimensional array, which itself is converted into a pointer if used as other than an lvalue. It follows from this that arrays are stored in row-major order (last subscript varies fastest).

4 EXAMPLE Consider the array object defined by the declaration

```
int x[3][5];
```

Here **x** is a 3×5 array of **int**s; more precisely, **x** is an array of three element objects, each of which is an array of five **int**s. In the expression **x[i]**, which is equivalent to **(*((x)+(i)))**, **x** is first converted to a pointer to the initial array of five **int**s. Then **i** is adjusted according to the type of **x**, which conceptually entails multiplying **i** by the size of the object to which the pointer points, namely an array of five **int** objects. The results are added and indirection is applied to yield an array of five **int**s. When used in the expression **x[i][j]**, that array is in turn converted to a pointer to the first of the **int**s, so **x[i][j]** yields an **int**.

Forward references: additive operators (6.5.6), address and indirection operators (6.5.3.2), array declarators (6.7.6.2).

6.5.2.2 Function calls

Constraints

1 The expression that denotes the called function[92)] shall have type pointer to function returning `void` or returning a complete object type other than an array type.

2 If the expression that denotes the called function has a type that includes a prototype, the number of arguments shall agree with the number of parameters. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

Semantics

3 A postfix expression followed by parentheses `()` containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function. The list of expressions specifies the arguments to the function.

4 An argument may be an expression of any complete object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[93)]

5 If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4. Otherwise, the function call has type `void`.

6 If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type `float` are promoted to `double`. These are called the *default argument promotions*. If the number of arguments does not equal the number of parameters, the behavior is undefined. If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (`, ...`) or the types of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined. If the function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, the behavior is undefined, except for the following cases:

— one promoted type is a signed integer type, the other promoted type is the corresponding unsigned integer type, and the value is representable in both types;

92) Most often, this is the result of converting an identifier that is a function designator.

93) A function may change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to. A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.

— both types are pointers to qualified or unqualified versions of a character type or **`void`**.

7 If the expression that denotes the called function has a type that does include a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type. The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments.

8 No other conversions are performed implicitly; in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a function prototype declarator.

9 If the function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function, the behavior is undefined.

10 There is a sequence point after the evaluations of the function designator and the actual arguments but before the actual call. Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.[94)]

11 Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions.

12 EXAMPLE In the function call

```
(*pf[f1()]) (f2(), f3() + f4())
```

the functions **`f1`**, **`f2`**, **`f3`**, and **`f4`** may be called in any order. All side effects have to be completed before the function pointed to by **`pf[f1()]`** is called.

Forward references: function declarators (including prototypes) (6.7.6.3), function definitions (6.9.1), the **`return`** statement (6.8.6.4), simple assignment (6.5.16.1).

6.5.2.3 Structure and union members

Constraints

1 The first operand of the **`.`** operator shall have an atomic, qualified, or unqualified structure or union type, and the second operand shall name a member of that type.

2 The first operand of the **`->`** operator shall have type ‘‘pointer to atomic, qualified, or unqualified structure’’ or ‘‘pointer to atomic, qualified, or unqualified union’’, and the second operand shall name a member of the type pointed to.

94) In other words, function executions do not ‘‘interleave’’ with each other.

Semantics

3 A postfix expression followed by the **.** operator and an identifier designates a member of a structure or union object. The value is that of the named member,[95] and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.

4 A postfix expression followed by the **->** operator and an identifier designates a member of a structure or union object. The value is that of the named member of the object to which the first expression points, and is an lvalue.[96] If the first expression is a pointer to a qualified type, the result has the so-qualified version of the type of the designated member.

5 Accessing a member of an atomic structure or union object results in undefined behavior.[97]

6 One special guarantee is made in order to simplify the use of unions: if a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the completed type of the union is visible. Two structures share a *common initial sequence* if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

7 EXAMPLE 1 If **f** is a function returning a structure or union, and **x** is a member of that structure or union, **f().x** is a valid postfix expression but is not an lvalue.

8 EXAMPLE 2 In:

```
struct s { int i; const int ci; };
struct s s;
const struct s cs;
volatile struct s vs;
```

the various members have the types:

---

95) If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called ‘‘type punning’’). This might be a trap representation.

96) If **&E** is a valid pointer expression (where **&** is the ‘‘address-of’’ operator, which generates a pointer to its operand), the expression **(&E)->MOS** is the same as **E.MOS**.

97) For example, a data race would occur if access to the entire structure or union in one thread conflicts with access to a member from another thread, where at least one access is a modification. Members can be safely accessed using a non-atomic object which is assigned to or from the atomic object.

```
s.i     int
s.ci    const int
cs.i    const int
cs.ci   const int
vs.i    volatile int
vs.ci   volatile const int
```

9 EXAMPLE 3 The following is a valid fragment:

```
union {
      struct {
            int    alltypes;
      } n;
      struct {
            int    type;
            int    intnode;
      } ni;
      struct {
            int    type;
            double doublenode;
      } nf;
} u;
u.nf.type = 1;
u.nf.doublenode = 3.14;
/* ... */
if (u.n.alltypes == 1)
      if (sin(u.nf.doublenode) == 0.0)
            /* ... */
```

The following is not a valid fragment (because the union type is not visible within function **f**):

```
struct t1 { int m; };
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.[98)]

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

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

6.5.2.5 Compound literals

Constraints

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

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

Semantics

3 A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a *compound literal*. It provides an unnamed object whose

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

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

with `old` being the result of the operation.

Special care must be taken if `E` has floating type; see 6.5.16.2.

value is given by the initializer list.[99)]

4 If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in 6.7.9, and the type of the compound literal is that of the completed array type. Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name. In either case, the result is an lvalue.

5 The value of the compound literal is that of an unnamed object initialized by the initializer list. If the compound literal occurs outside the body of a function, the object has static storage duration; otherwise, it has automatic storage duration associated with the enclosing block.

6 All the semantic rules for initializer lists in 6.7.9 also apply to compound literals.[100)]

7 String literals, and compound literals with const-qualified types, need not designate distinct objects.[101)]

8 EXAMPLE 1 The file scope definition

```
int *p = (int []){2, 4};
```

initializes **p** to point to the first element of an array of two ints, the first having the value two and the second, four. The expressions in this compound literal are required to be constant. The unnamed object has static storage duration.

9 EXAMPLE 2 In contrast, in

```
void f(void)
{
      int *p;
      /*...*/
      p = (int [2]){*p};
      /*...*/
}
```

p is assigned the address of the first element of an array of two ints, the first having the value previously pointed to by **p** and the second, zero. The expressions in this compound literal need not be constant. The unnamed object has automatic storage duration.

10 EXAMPLE 3 Initializers with designations can be combined with compound literals. Structure objects created using compound literals can be passed to functions without depending on member order:

```
drawline((struct point){.x=1, .y=1},
      (struct point){.x=3, .y=4});
```

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

100) For example, subobjects without explicit initializers are initialized to zero.

101) This allows implementations to share storage for string literals and constant compound literals with the same or overlapping representations.

Or, if **drawline** instead expected pointers to **struct point**:

```
drawline(&(struct point){.x=1, .y=1},
      &(struct point){.x=3, .y=4});
```

11 EXAMPLE 4 A read-only compound literal can be specified through constructions like:

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

12 EXAMPLE 5 The following three expressions have different meanings:

```
"/tmp/fileXXXXXX"
(char []){"/tmp/fileXXXXXX"}
(const char []){"/tmp/fileXXXXXX"}
```

The first always has static storage duration and has type array of **char**, but need not be modifiable; the last two have automatic storage duration when they occur within the body of a function, and the first of these two is modifiable.

13 EXAMPLE 6 Like string literals, const-qualified compound literals can be placed into read-only memory and can even be shared. For example,

```
(const char []){"abc"} == "abc"
```

might yield 1 if the literals' storage is shared.

14 EXAMPLE 7 Since compound literals are unnamed, a single compound literal cannot specify a circularly linked object. For example, there is no way to write a self-referential compound literal that could be used as the function argument in place of the named object **endless_zeros** below:

```
struct int_list { int car; struct int_list *cdr; };
struct int_list endless_zeros = {0, &endless_zeros};
eval(endless_zeros);
```

15 EXAMPLE 8 Each compound literal creates only a single object in a given scope:

```
struct s { int i; };

int f (void)
{
      struct s *p = 0, *q;
      int j = 0;

again:
      q = p, p = &((struct s){ j++ });
      if (j < 2) goto again;

      return p == q && q->i == 1;
}
```

The function **f()** always returns the value 1.

16 Note that if an iteration statement were used instead of an explicit **goto** and a labeled statement, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around **p** would have an indeterminate value, which would result in undefined behavior.

Forward references: type names (6.7.7), initialization (6.7.9).

6.5.3 Unary operators

Syntax

1 *unary-expression:*
  postfix-expression
  ++ *unary-expression*
  -- *unary-expression*
  unary-operator cast-expression
  sizeof *unary-expression*
  sizeof (*type-name* **)**
  _Alignof (*type-name* **)**

unary-operator: one of
  & * + - ~ !

6.5.3.1 Prefix increment and decrement operators

Constraints

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

Semantics

2 The value of the operand of the prefix **++** operator is incremented. The result is the new value of the operand after incrementation. The expression **++E** is equivalent to **(E+=1)**. See the discussions of additive operators and compound assignment for information on constraints, types, side effects, and conversions and the effects of operations on pointers.

3 The prefix **--** operator is analogous to the prefix **++** operator, except that the value of the operand is decremented.

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

6.5.3.2 Address and indirection operators

Constraints

1 The operand of the unary **&** operator shall be either a function designator, the result of a **[]** or unary ***** operator, or an lvalue that designates an object that is not a bit-field and is not declared with the **register** storage-class specifier.

2 The operand of the unary ***** operator shall have pointer type.

Semantics

3 The unary **&** operator yields the address of its operand. If the operand has type ‘‘*type*’’, the result has type ‘‘pointer to *type*’’. If the operand is the result of a unary ***** operator, neither that operator nor the **&** operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an

lvalue. Similarly, if the operand is the result of a **[]** operator, neither the **&** operator nor the unary ***** that is implied by the **[]** is evaluated and the result is as if the **&** operator were removed and the **[]** operator were changed to a **+** operator. Otherwise, the result is a pointer to the object or function designated by its operand.

4 The unary ***** operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type ‘‘pointer to *type*’’, the result has type ‘‘*type*’’. If an invalid value has been assigned to the pointer, the behavior of the unary ***** operator is undefined.[102)]

Forward references: storage-class specifiers (6.7.1), structure and union specifiers (6.7.2.1).

6.5.3.3 Unary arithmetic operators

Constraints

1 The operand of the unary **+** or **-** operator shall have arithmetic type; of the **~** operator, integer type; of the **!** operator, scalar type.

Semantics

2 The result of the unary **+** operator is the value of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.

3 The result of the unary **-** operator is the negative of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.

4 The result of the **~** operator is the bitwise complement of its (promoted) operand (that is, each bit in the result is set if and only if the corresponding bit in the converted operand is not set). The integer promotions are performed on the operand, and the result has the promoted type. If the promoted type is an unsigned type, the expression **~E** is equivalent to the maximum value representable in that type minus **E**.

5 The result of the logical negation operator **!** is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0. The result has type **int**. The expression **!E** is equivalent to **(0==E)**.

---

102) Thus, **&*E** is equivalent to **E** (even if **E** is a null pointer), and **&(E1[E2])** to **((E1)+(E2))**. It is always true that if **E** is a function designator or an lvalue that is a valid operand of the unary **&** operator, ***&E** is a function designator or an lvalue equal to **E**. If ***P** is an lvalue and **T** is the name of an object pointer type, ***(T)P** is an lvalue that has a type compatible with that to which **T** points.

Among the invalid values for dereferencing a pointer by the unary ***** operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime.

6.5.3.4 The `sizeof` and `_Alignof` operators

Constraints

1 The **`sizeof`** operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member. The **`_Alignof`** operator shall not be applied to a function type or an incomplete type.

Semantics

2 The **`sizeof`** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.

3 The **`_Alignof`** operator yields the alignment requirement of its operand type. The operand is not evaluated and the result is an integer constant. When applied to an array type, the result is the alignment requirement of the element type.

4 When **`sizeof`** is applied to an operand that has type **`char`**, **`unsigned char`**, or **`signed char`**, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array.[103] When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding.

5 The value of the result of both operators is implementation-defined, and its type (an unsigned integer type) is **`size_t`**, defined in **`<stddef.h>`** (and other headers).

6 EXAMPLE 1 A principal use of the **`sizeof`** operator is in communication with routines such as storage allocators and I/O systems. A storage-allocation function might accept a size (in bytes) of an object to allocate and return a pointer to **`void`**. For example:

```
extern void *alloc(size_t);
double *dp = alloc(sizeof *dp);
```

The implementation of the **`alloc`** function should ensure that its return value is aligned suitably for conversion to a pointer to **`double`**.

7 EXAMPLE 2 Another use of the **`sizeof`** operator is to compute the number of elements in an array:

```
sizeof array / sizeof array[0]
```

8 EXAMPLE 3 In this example, the size of a variable length array is computed and returned from a function:

```
#include <stddef.h>
```

103) When applied to a parameter declared to have array or function type, the **`sizeof`** operator yields the size of the adjusted (pointer) type (see 6.9.1).

```
size_t fsize3(int n)
{
      char b[n+3];        // variable length array
      return sizeof b;    // execution time sizeof
}

int main()
{
      size_t size;
      size = fsize3(10); // fsize3 returns 13
      return 0;
}
```

Forward references: common definitions `<stddef.h>` (7.19), declarations (6.7), structure and union specifiers (6.7.2.1), type names (6.7.7), array declarators (6.7.6.2).

6.5.4 Cast operators

Syntax

1 *cast-expression:*
 unary-expression
 `(` *type-name* `)` *cast-expression*

Constraints

2 Unless the type name specifies a void type, the type name shall specify atomic, qualified, or unqualified scalar type, and the operand shall have scalar type.

3 Conversions that involve pointers, other than where permitted by the constraints of 6.5.16.1, shall be specified by means of an explicit cast.

4 A pointer type shall not be converted to any floating type. A floating type shall not be converted to any pointer type.

Semantics

5 Preceding an expression by a parenthesized type name converts the value of the expression to the named type. This construction is called a *cast*.[104] A cast that specifies no conversion has no effect on the type or value of an expression.

6 If the value of the expression is represented with greater range or precision than required by the type named by the cast (6.3.1.8), then the cast specifies a conversion even if the type of the expression is the same as the named type and removes any extra range and precision.

Forward references: equality operators (6.5.9), function declarators (including prototypes) (6.7.6.3), simple assignment (6.5.16.1), type names (6.7.7).

104) A cast does not yield an lvalue. Thus, a cast to a qualified type has the same effect as a cast to the unqualified version of the type.

6.5.5 Multiplicative operators

Syntax

1 *multiplicative-expression:*
  cast-expression
  multiplicative-expression **`*`** *cast-expression*
  multiplicative-expression **`/`** *cast-expression*
  multiplicative-expression **`%`** *cast-expression*

Constraints

2 Each of the operands shall have arithmetic type. The operands of the **`%`** operator shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the binary **`*`** operator is the product of the operands.

5 The result of the **`/`** operator is the quotient from the division of the first operand by the second; the result of the **`%`** operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.

6 When integers are divided, the result of the **`/`** operator is the algebraic quotient with any fractional part discarded.[105] If the quotient **`a/b`** is representable, the expression **`(a/b)*b + a%b`** shall equal **`a`**; otherwise, the behavior of both **`a/b`** and **`a%b`** is undefined.

6.5.6 Additive operators

Syntax

1 *additive-expression:*
  multiplicative-expression
  additive-expression **`+`** *multiplicative-expression*
  additive-expression **`-`** *multiplicative-expression*

Constraints

2 For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to a complete object type and the other shall have integer type. (Incrementing is equivalent to adding 1.)

3 For subtraction, one of the following shall hold:

105) This is often called ‘‘truncation toward zero’’.

— both operands have arithmetic type;

— both operands are pointers to qualified or unqualified versions of compatible complete object types; or

— the left operand is a pointer to a complete object type and the right operand has integer type.

(Decrementing is equivalent to subtracting 1.)

Semantics

4 If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

5 The result of the binary **+** operator is the sum of the operands.

6 The result of the binary **-** operator is the difference resulting from the subtraction of the second operand from the first.

7 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

8 When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. In other words, if the expression **P** points to the *i*-th element of an array object, the expressions **(P)+N** (equivalently, **N+(P)**) and **(P)-N** (where **N** has the value *n*) point to, respectively, the $i+n$-th and $i-n$-th elements of the array object, provided they exist. Moreover, if the expression **P** points to the last element of an array object, the expression **(P)+1** points one past the last element of the array object, and if the expression **Q** points one past the last element of an array object, the expression **(Q)-1** points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary ***** operator that is evaluated.

9 When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is **ptrdiff_t** defined in the **<stddef.h>** header. If the result is not representable in an object of that type, the behavior is undefined. In other words, if the expressions **P** and **Q** point to, respectively, the *i*-th and *j*-th elements of an array object, the expression **(P)-(Q)** has the value $i-j$ provided the value fits in an

object of type **`ptrdiff_t`**. Moreover, if the expression **`P`** points either to an element of an array object or one past the last element of an array object, and the expression **`Q`** points to the last element of the same array object, the expression **`((Q)+1)-(P)`** has the same value as **`((Q)-(P))+1`** and as **`-((P)-((Q)+1))`**, and has the value zero if the expression **`P`** points one past the last element of the array object, even though the expression **`(Q)+1`** does not point to an element of the array object.[106)]

10 EXAMPLE Pointer arithmetic is well defined with pointers to variable length array types.

```
{
        int n = 4, m = 3;
        int a[n][m];
        int (*p)[m] = a;  // p == &a[0]
        p += 1;           // p == &a[1]
        (*p)[2] = 99;     // a[1][2] == 99
        n = p - a;        // n == 1
}
```

11 If array **`a`** in the above example were declared to be an array of known constant size, and pointer **`p`** were declared to be a pointer to an array of the same known constant size (pointing to **`a`**), the results would be the same.

Forward references: array declarators (6.7.6.2), common definitions **`<stddef.h>`** (7.19).

6.5.7 Bitwise shift operators

Syntax

1 *shift-expression:*
  additive-expression
  shift-expression **`<<`** *additive-expression*
  shift-expression **`>>`** *additive-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is

106) Another way to approach pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the integer expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type. For pointer subtraction, the result of the difference between the character pointers is similarly divided by the size of the object originally pointed to.

When viewed in this way, an implementation need only provide one extra byte (which may overlap another object in the program) just after the end of the object in order to satisfy the ''one past the last element'' requirements.

greater than or equal to the width of the promoted left operand, the behavior is undefined.

4 The result of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros. If `E1` has an unsigned type, the value of the result is $\texttt{E1} \times 2^{\texttt{E2}}$, reduced modulo one more than the maximum value representable in the result type. If `E1` has a signed type and nonnegative value, and $\texttt{E1} \times 2^{\texttt{E2}}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

5 The result of `E1 >> E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of $\texttt{E1} / 2^{\texttt{E2}}$. If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

6.5.8 Relational operators

Syntax

1 *relational-expression:*
shift-expression
relational-expression `<` *shift-expression*
relational-expression `>` *shift-expression*
relational-expression `<=` *shift-expression*
relational-expression `>=` *shift-expression*

Constraints

2 One of the following shall hold:

— both operands have real type; or

— both operands are pointers to qualified or unqualified versions of compatible object types.

Semantics

3 If both of the operands have arithmetic type, the usual arithmetic conversions are performed.

4 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

5 When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to. If two pointers to object types both point to the same object, or both point one past the last element of the same array object, they compare equal. If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array with lower subscript

values. All pointers to members of the same union object compare equal. If the expression `P` points to an element of an array object and the expression `Q` points to the last element of the same array object, the pointer expression `Q+1` compares greater than `P`. In all other cases, the behavior is undefined.

6 Each of the operators `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) shall yield 1 if the specified relation is true and 0 if it is false.[107] The result has type `int`.

6.5.9 Equality operators

Syntax

1 *equality-expression:*
relational-expression
equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

Constraints

2 One of the following shall hold:

— both operands have arithmetic type;

— both operands are pointers to qualified or unqualified versions of compatible types;

— one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of `void`; or

— one operand is a pointer and the other is a null pointer constant.

Semantics

3 The `==` (equal to) and `!=` (not equal to) operators are analogous to the relational operators except for their lower precedence.[108] Each of the operators yields 1 if the specified relation is true and 0 if it is false. The result has type `int`. For any pair of operands, exactly one of the relations is true.

4 If both of the operands have arithmetic type, the usual arithmetic conversions are performed. Values of complex types are equal if and only if both their real parts are equal and also their imaginary parts are equal. Any two values of arithmetic types from different type domains are equal if and only if the results of their conversions to the (complex) result type determined by the usual arithmetic conversions are equal.

---

107) The expression `a<b<c` is not interpreted as in ordinary mathematics. As the syntax indicates, it means `(a<b)<c`; in other words, ‘‘if `a` is less than `b`, compare 1 to `c`; otherwise, compare 0 to `c`’’.

108) Because of the precedences, `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth-value.

5 Otherwise, at least one operand is a pointer. If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer. If one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of **`void`**, the former is converted to the type of the latter.

6 Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.[109)]

7 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

6.5.10 Bitwise AND operator

Syntax

1 *AND-expression:*
equality-expression
AND-expression **`&`** *equality-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the binary **`&`** operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set).

109) Two objects may be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior.

6.5.11 Bitwise exclusive OR operator

Syntax

1 *exclusive-OR-expression:*
  AND-expression
  exclusive-OR-expression `^` *AND-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the `^` operator is the bitwise exclusive OR of the operands (that is, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set).

6.5.12 Bitwise inclusive OR operator

Syntax

1 *inclusive-OR-expression:*
  exclusive-OR-expression
  inclusive-OR-expression `|` *exclusive-OR-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the `|` operator is the bitwise inclusive OR of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set).

6.5.13 Logical AND operator

Syntax

1 *logical-AND-expression:*
  inclusive-OR-expression
  logical-AND-expression **&&** *inclusive-OR-expression*

Constraints

2 Each of the operands shall have scalar type.

Semantics

3 The **&&** operator shall yield 1 if both of its operands compare unequal to 0; otherwise, it yields 0. The result has type **int**.

4 Unlike the bitwise binary **&** operator, the **&&** operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares equal to 0, the second operand is not evaluated.

6.5.14 Logical OR operator

Syntax

1 *logical-OR-expression:*
  logical-AND-expression
  logical-OR-expression || *logical-AND-expression*

Constraints

2 Each of the operands shall have scalar type.

Semantics

3 The || operator shall yield 1 if either of its operands compare unequal to 0; otherwise, it yields 0. The result has type **int**.

4 Unlike the bitwise | operator, the || operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares unequal to 0, the second operand is not evaluated.

6.5.15 Conditional operator

Syntax

1 *conditional-expression:*
logical-OR-expression
logical-OR-expression **`?`** *expression* **`:`** *conditional-expression*

Constraints

2 The first operand shall have scalar type.

3 One of the following shall hold for the second and third operands:

— both operands have arithmetic type;

— both operands have the same structure or union type;

— both operands have void type;

— both operands are pointers to qualified or unqualified versions of compatible types;

— one operand is a pointer and the other is a null pointer constant; or

— one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of **`void`**.

Semantics

4 The first operand is evaluated; there is a sequence point between its evaluation and the evaluation of the second or third operand (whichever is evaluated). The second operand is evaluated only if the first compares unequal to 0; the third operand is evaluated only if the first compares equal to 0; the result is the value of the second or third operand (whichever is evaluated), converted to the type described below.[110)]

5 If both the second and third operands have arithmetic type, the result type that would be determined by the usual arithmetic conversions, were they applied to those two operands, is the type of the result. If both the operands have structure or union type, the result has that type. If both operands have void type, the result has void type.

6 If both the second and third operands are pointers or one is a null pointer constant and the other is a pointer, the result type is a pointer to a type qualified with all the type qualifiers of the types referenced by both operands. Furthermore, if both operands are pointers to compatible types or to differently qualified versions of compatible types, the result type is a pointer to an appropriately qualified version of the composite type; if one operand is a null pointer constant, the result has the type of the other operand; otherwise, one operand is a pointer to **`void`** or a qualified version of **`void`**, in which case the result type is a pointer to an appropriately qualified version of **`void`**.

---

110) A conditional expression does not yield an lvalue.

7 EXAMPLE The common type that results when the second and third operands are pointers is determined in two independent stages. The appropriate qualifiers, for example, do not depend on whether the two pointers have compatible types.

8 Given the declarations

```
const void *c_vp;
void *vp;
const int *c_ip;
volatile int *v_ip;
int *ip;
const char *c_cp;
```

the third column in the following table is the common type that is the result of a conditional expression in which the first two columns are the second and third operands (in either order):

```
c_vp   c_ip   const void *
v_ip   0      volatile int *
c_ip   v_ip   const volatile int *
vp     c_cp   const void *
ip     c_ip   const int *
vp     ip     void *
```

6.5.16 Assignment operators

Syntax

1 *assignment-expression:*
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of
= *= /= %= += -= <<= >>= &= ^= |=

Constraints

2 An assignment operator shall have a modifiable lvalue as its left operand.

Semantics

3 An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment,[111] but is not an lvalue. The type of an assignment expression is the type the left operand would have after lvalue conversion. The side effect of updating the stored value of the left operand is sequenced after the value computations of the left and right operands. The evaluations of the operands are unsequenced.

111) The implementation is permitted to read the object to determine the value but is not required to, even when the object has volatile-qualified type.

6.5.16.1 Simple assignment

Constraints

1 One of the following shall hold:[112)]

— the left operand has atomic, qualified, or unqualified arithmetic type, and the right has arithmetic type;

— the left operand has an atomic, qualified, or unqualified version of a structure or union type compatible with the type of the right;

— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) one operand is a pointer to an object type, and the other is a pointer to a qualified or unqualified version of **`void`**, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

— the left operand is an atomic, qualified, or unqualified pointer, and the right is a null pointer constant; or

— the left operand has type atomic, qualified, or unqualified **`_Bool`**, and the right is a pointer.

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

112) The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in 6.3.2.1) that changes lvalues to ‘‘the value of the expression’’ and thus removes any type qualifiers that were applied to the type category of the expression (for example, it removes **`const`** but not **`volatile`** from the type **`int volatile * const`**).

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

the `int` value returned by the function may 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` should 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.

Semantics

3 A *compound assignment* of the form `E1` *op*`=` `E2` is equivalent to the simple assignment expression `E1 = E1` *op* `(E2)`, except that the lvalue `E1` is evaluated only once, and with respect to an indeterminately-sequenced function call, the operation of a compound

assignment is a single evaluation. If **E1** has an atomic type, compound assignment is a read-modify-write operation with **memory_order_seq_cst** memory order semantics.113)

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

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

with **new** being the result of the operation.

If **E1** or **E2** has floating type, then exceptional conditions or floating-point exceptions encountered during discarded evaluations of **new** should 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;
      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* must 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*

Semantics

2 The left operand of a comma operator is evaluated as a void expression; there is a sequence point between its evaluation and that of the right operand. Then the right operand is evaluated; the result has its type and value.[114)]

3 EXAMPLE As indicated by the syntax, the comma operator (as described in this subclause) cannot appear in contexts where a comma is used to separate items in a list (such as arguments to functions or lists of initializers). On the other hand, it can be used within a parenthesized expression or within the second expression of a conditional operator in such contexts. In the function call

```
f(a, (t=3, t+2), c)
```

the function has three arguments, the second of which has the value 5.

Forward references: initialization (6.7.9).

114) A comma operator does not yield an lvalue.

6.6 Constant expressions

Syntax

1 *constant-expression:*
conditional-expression

Description

2 A constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be.

Constraints

3 Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated.[115)]

4 Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

Semantics

5 An expression that evaluates to a constant is required in several contexts. If a floating expression is evaluated in the translation environment, the arithmetic range and precision shall be at least as great as if the expression were being evaluated in the execution environment.[116)]

6 An *integer constant expression*[117)] shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, **`sizeof`** expressions whose results are integer constants, **`_Alignof`** expressions, and floating constants that are the immediate operands of casts. Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the **`sizeof`** or **`_Alignof`** operator.

7 More latitude is permitted for constant expressions in initializers. Such a constant expression shall be, or evaluate to, one of the following:

— an arithmetic constant expression,

---

115) The operand of a **`sizeof`** or **`_Alignof`** operator is usually not evaluated (6.5.3.4).

116) The use of evaluation formats as characterized by **`FLT_EVAL_METHOD`** also applies to evaluation in the translation environment.

117) An integer constant expression is required in a number of contexts such as the size of a bit-field member of a structure, the value of an enumeration constant, and the size of a non-variable length array. Further constraints that apply to the integer constant expressions used in conditional-inclusion preprocessing directives are discussed in 6.10.1.

— a null pointer constant,

— an address constant, or

— an address constant for a complete object type plus or minus an integer constant expression.

8 An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, **`sizeof`** expressions whose results are integer constants, and **`_Alignof`** expressions. Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to a **`sizeof`** or **`_Alignof`** operator.

9 An *address constant* is a null pointer, a pointer to an lvalue designating an object of static storage duration, or a pointer to a function designator; it shall be created explicitly using the unary **`&`** operator or an integer constant cast to pointer type, or implicitly by the use of an expression of array or function type. The array-subscript **`[]`** and member-access **`.`** and **`->`** operators, the address **`&`** and indirection **`*`** unary operators, and pointer casts may be used in the creation of an address constant, but the value of an object shall not be accessed by use of these operators.

10 An implementation may accept other forms of constant expressions.

11 The semantic rules for the evaluation of a constant expression are the same as for nonconstant expressions.[118)]

Forward references: array declarators (6.7.6.2), initialization (6.7.9).

---

118) Thus, in the following initialization,

```
static int i = 2 || 1 / 0;
```

the expression is a valid integer constant expression with value one.

6.7 Declarations

Syntax

1 *declaration:*
    declaration-specifiers init-declarator-list$_{opt}$ **;**
    static_assert-declaration

declaration-specifiers:
    storage-class-specifier declaration-specifiers$_{opt}$
    type-specifier declaration-specifiers$_{opt}$
    type-qualifier declaration-specifiers$_{opt}$
    function-specifier declaration-specifiers$_{opt}$
    alignment-specifier declaration-specifiers$_{opt}$

init-declarator-list:
    init-declarator
    init-declarator-list **,** *init-declarator*

init-declarator:
    declarator
    declarator **=** *initializer*

Constraints

2 A declaration other than a static_assert declaration shall declare at least a declarator (other than the parameters of a function or the members of a structure or union), a tag, or the members of an enumeration.

3 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except that:

— a typedef name may be redefined to denote the same type as it currently does, provided that type is not a variably modified type;

— tags may be redeclared as specified in 6.7.2.3.

4 All declarations in the same scope that refer to the same object or function shall specify compatible types.

Semantics

5 A declaration specifies the interpretation and attributes of a set of identifiers. A *definition* of an identifier is a declaration for that identifier that:

— for an object, causes storage to be reserved for that object;

— for a function, includes the function body;[119)]

— for an enumeration constant, is the (only) declaration of the identifier;

— for a typedef name, is the first (or only) declaration of the identifier.

6 The declaration specifiers consist of a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. The init-declarator-list is a comma-separated sequence of declarators, each of which may have additional type information, or an initializer, or both. The declarators contain the identifiers (if any) being declared.

7 If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer; in the case of function parameters (including in prototypes), it is the adjusted type (see 6.7.6.3) that is required to be complete.

Forward references: declarators (6.7.6), enumeration specifiers (6.7.2.2), initialization (6.7.9), type names (6.7.7), type qualifiers (6.7.3).

6.7.1 Storage-class specifiers

Syntax

1 *storage-class-specifier:*

```
typedef
extern
static
_Thread_local
auto
register
```

Constraints

2 At most, one storage-class specifier may be given in the declaration specifiers in a declaration, except that **`_Thread_local`** may appear with **`static`** or **`extern`**.[120)]

3 In the declaration of an object with block scope, if the declaration specifiers include **`_Thread_local`**, they shall also include either **`static`** or **`extern`**. If **`_Thread_local`** appears in any declaration of an object, it shall be present in every declaration of that object.

4 **`_Thread_local`** shall not appear in the declaration specifiers of a function declaration.

119) Function definitions have a different syntax, described in 6.9.1.

120) See ‘‘future language directions’’ (6.11.5).

Semantics

5 The **`typedef`** specifier is called a ‘‘storage-class specifier’’ for syntactic convenience only; it is discussed in 6.7.8. The meanings of the various linkages and storage durations were discussed in 6.2.2 and 6.2.4.

6 A declaration of an identifier for an object with storage-class specifier **`register`** suggests that access to the object be as fast as possible. The extent to which such suggestions are effective is implementation-defined.[121)]

7 The declaration of an identifier for a function that has block scope shall have no explicit storage-class specifier other than **`extern`**.

8 If an aggregate or union object is declared with a storage-class specifier other than **`typedef`**, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the members of the object, and so on recursively for any aggregate or union member objects.

Forward references: type definitions (6.7.8).

121) The implementation may treat any **`register`** declaration simply as an **`auto`** declaration. However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier **`register`** cannot be computed, either explicitly (by use of the unary **`&`** operator as discussed in 6.5.3.2) or implicitly (by converting an array name to a pointer as discussed in 6.3.2.1). Thus, the only operators that can be applied to an array declared with storage-class specifier **`register`** are **`sizeof`** and **`_Alignof`**.

6.7.2 Type specifiers

Syntax

1 *type-specifier:*
`void`
`char`
`short`
`int`
`long`
`float`
`double`
`signed`
`unsigned`
`_Bool`
`_Complex`
atomic-type-specifier
struct-or-union-specifier
enum-specifier
typedef-name

Constraints

2 At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each struct declaration and type name. Each list of type specifiers shall be one of the following multisets (delimited by commas, when there is more than one multiset per item); the type specifiers may occur in any order, possibly intermixed with the other declaration specifiers.

— **`void`**

— **`char`**

— **`signed char`**

— **`unsigned char`**

— **`short`**, **`signed short`**, **`short int`**, or **`signed short int`**

— **`unsigned short`**, or **`unsigned short int`**

— **`int`**, **`signed`**, or **`signed int`**

— **`unsigned`**, or **`unsigned int`**

— **`long`**, **`signed long`**, **`long int`**, or **`signed long int`**

— **`unsigned long`**, or **`unsigned long int`**

— **`long long`**, **`signed long long`**, **`long long int`**, or **`signed long long int`**

— **`unsigned long long`**, or **`unsigned long long int`**

— **`float`**

— **`double`**

— **`long double`**

— **`_Bool`**

— **`float _Complex`**

— **`double _Complex`**

— **`long double _Complex`**

— atomic type specifier

— struct or union specifier

— enum specifier

— typedef name

3 The type specifier **`_Complex`** shall not be used if the implementation does not support complex types (see 6.10.8.3).

Semantics

4 Specifiers for structures, unions, enumerations, and atomic types are discussed in 6.7.2.1 through 6.7.2.4. Declarations of typedef names are discussed in 6.7.8. The characteristics of the other types are discussed in 6.2.5.

5 Each of the comma-separated multisets designates the same type, except that for bit-fields, it is implementation-defined whether the specifier **`int`** designates the same type as **`signed int`** or the same type as **`unsigned int`**.

Forward references: atomic type specifiers (6.7.2.4), enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1), tags (6.7.2.3), type definitions (6.7.8).

6.7.2.1 Structure and union specifiers

Syntax

1 *struct-or-union-specifier:*
struct-or-union identifier$_{opt}$ **{** *struct-declaration-list* **}**
struct-or-union identifier

struct-or-union:
  `struct`
  `union`

struct-declaration-list:
  struct-declaration
  struct-declaration-list struct-declaration

struct-declaration:
  specifier-qualifier-list struct-declarator-list$_{opt}$ **`;`**
  static_assert-declaration

specifier-qualifier-list:
  type-specifier specifier-qualifier-list$_{opt}$
  type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
  struct-declarator
  struct-declarator-list **`,`** *struct-declarator*

struct-declarator:
  declarator
  declarator$_{opt}$ **`:`** *constant-expression*

Constraints

2 A struct-declaration that does not declare an anonymous structure or anonymous union shall contain a struct-declarator-list.

3 A structure or union shall not contain a member with incomplete or function type (hence, a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself), except that the last member of a structure with more than one named member may have incomplete array type; such a structure (and any union containing, possibly recursively, a member that is such a structure) shall not be a member of a structure or an element of an array.

4 The expression that specifies the width of a bit-field shall be an integer constant expression with a nonnegative value that does not exceed the width of an object of the type that would be specified were the colon and expression omitted.[122] If the value is zero, the declaration shall have no declarator.

5 A bit-field shall have a type that is a qualified or unqualified version of **`_Bool`**, **`signed int`**, **`unsigned int`**, or some other implementation-defined type. It is implementation-defined whether atomic types are permitted.

---

122) While the number of bits in a **`_Bool`** object is at least **`CHAR_BIT`**, the width (number of sign and value bits) of a **`_Bool`** may be just 1 bit.

Semantics

6 As discussed in 6.2.5, a structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of members whose storage overlap.

7 Structure and union specifiers have the same form. The keywords **`struct`** and **`union`** indicate that the type being specified is, respectively, a structure type or a union type.

8 The presence of a struct-declaration-list in a struct-or-union-specifier declares a new type, within a translation unit. The struct-declaration-list is a sequence of declarations for the members of the structure or union. If the struct-declaration-list does not contain any named members, either directly or via an anonymous structure or anonymous union, the behavior is undefined. The type is incomplete until immediately after the `}` that terminates the list, and complete thereafter.

9 A member of a structure or union may have any complete object type other than a variably modified type.[123)] In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a *bit-field*;[124)] its width is preceded by a colon.

10 A bit-field is interpreted as having a signed or unsigned integer type consisting of the specified number of bits.[125)] If the value 0 or 1 is stored into a nonzero-width bit-field of type **`_Bool`**, the value of the bit-field shall compare equal to the value stored; a **`_Bool`** bit-field has the semantics of a **`_Bool`**.

11 An implementation may allocate any addressable storage unit large enough to hold a bit-field. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. The alignment of the addressable storage unit is unspecified.

12 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[126)] As a special case, a bit-field structure member with a width of 0

---

123) A structure or union cannot contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3.

124) The unary **`&`** (address-of) operator cannot be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects.

125) As specified in 6.7.2 above, if the actual type specifier used is **`int`** or a typedef-name defined as **`int`**, then it is implementation-defined whether the bit-field is signed or unsigned.

126) An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts.

indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

13 An unnamed member whose type specifier is a structure specifier with no tag is called an *anonymous structure*; an unnamed member whose type specifier is a union specifier with no tag is called an *anonymous union*. The members of an anonymous structure or union are considered to be members of the containing structure or union. This applies recursively if the containing structure or union is also anonymous.

14 Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

15 Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may be unnamed padding within a structure object, but not at its beginning.

16 The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa.

17 There may be unnamed padding at the end of a structure or union.

18 As a special case, the last element of a structure with more than one named member may have an incomplete array type; this is called a *flexible array member*. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply. However, when a `.` (or `->`) operator has a left operand that is (a pointer to) a structure with a flexible array member and the right operand names that member, it behaves as if that member were replaced with the longest array (with the same element type) that would not make the structure larger than the object being accessed; the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array. If this array would have no elements, it behaves as if it had one element but the behavior is undefined if any attempt is made to access that element or to generate a pointer one past it.

19 EXAMPLE 1 The following illustrates anonymous structures and unions:

```
struct v {
      union {      // anonymous union
            struct { int i, j; };    // anonymous structure
            struct { long k, l; } w;
      };
      int m;
} v1;
```

```
v1.i = 2;   // valid
v1.k = 3;   // invalid: inner structure is not anonymous
v1.w.k = 5; // valid
```

20 EXAMPLE 2 After the declaration:

```
struct s { int n; double d[]; };
```

the structure **struct s** has a flexible array member **d**. A typical way to use this is:

```
int m = /* some value */;
struct s *p = malloc(sizeof (struct s) + sizeof (double [m]));
```

and assuming that the call to **malloc** succeeds, the object pointed to by **p** behaves, for most purposes, as if **p** had been declared as:

```
struct { int n; double d[m]; } *p;
```

(there are circumstances in which this equivalence is broken; in particular, the offsets of member **d** might not be the same).

21 Following the above declaration:

```
struct s t1 = { 0 };            // valid
struct s t2 = { 1, { 4.2 }};    // invalid
t1.n = 4;                       // valid
t1.d[0] = 4.2;                  // might be undefined behavior
```

The initialization of **t2** is invalid (and violates a constraint) because **struct s** is treated as if it did not contain member **d**. The assignment to **t1.d[0]** is probably undefined behavior, but it is possible that

```
sizeof (struct s) >= offsetof(struct s, d) + sizeof (double)
```

in which case the assignment would be legitimate. Nevertheless, it cannot appear in strictly conforming code.

22 After the further declaration:

```
struct ss { int n; };
```

the expressions:

```
sizeof (struct s) >= sizeof (struct ss)
sizeof (struct s) >= offsetof(struct s, d)
```

are always equal to 1.

23 If **sizeof (double)** is 8, then after the following code is executed:

```
struct s *s1;
struct s *s2;
s1 = malloc(sizeof (struct s) + 64);
s2 = malloc(sizeof (struct s) + 46);
```

and assuming that the calls to **malloc** succeed, the objects pointed to by **s1** and **s2** behave, for most purposes, as if the identifiers had been declared as:

```
struct { int n; double d[8]; } *s1;
struct { int n; double d[5]; } *s2;
```

24 Following the further successful assignments:

```
s1 = malloc(sizeof (struct s) + 10);
s2 = malloc(sizeof (struct s) +  6);
```

they then behave as if the declarations were:

```
struct { int n; double d[1]; } *s1, *s2;
```

and:

```
double *dp;
dp = &(s1->d[0]); // valid
*dp = 42;         // valid
dp = &(s2->d[0]); // valid
*dp = 42;         // undefined behavior
```

25 The assignment:

```
*s1 = *s2;
```

only copies the member **n**; if any of the array elements are within the first **sizeof (struct s)** bytes of the structure, they might be copied or simply overwritten with indeterminate values.

26 EXAMPLE 3 Because members of anonymous structures and unions are considered to be members of the containing structure or union, **struct s** in the following example has more than one named member and thus the use of a flexible array member is valid:

```
struct s {
      struct { int i; };
      int a[];
};
```

Forward references: declarators (6.7.6), tags (6.7.2.3).

6.7.2.2 Enumeration specifiers

Syntax

1 *enum-specifier:*
  enum *identifier*$_{opt}$ **{** *enumerator-list* **}**
  enum *identifier*$_{opt}$ **{** *enumerator-list* **, }**
  enum *identifier*

enumerator-list:
  enumerator
  enumerator-list **,** *enumerator*

enumerator:
  enumeration-constant
  enumeration-constant **=** *constant-expression*

Constraints

2 The expression that defines the value of an enumeration constant shall be an integer constant expression that has a value representable as an **int**.

Semantics

3 The identifiers in an enumerator list are declared as constants that have type **`int`** and may appear wherever such are permitted.[127)] An enumerator with **`=`** defines its enumeration constant as the value of the constant expression. If the first enumerator has no **`=`**, the value of its enumeration constant is 0. Each subsequent enumerator with no **`=`** defines its enumeration constant as the value of the constant expression obtained by adding 1 to the value of the previous enumeration constant. (The use of enumerators with **`=`** may produce enumeration constants with values that duplicate other values in the same enumeration.) The enumerators of an enumeration are also known as its members.

4 Each enumerated type shall be compatible with **`char`**, a signed integer type, or an unsigned integer type. The choice of type is implementation-defined,[128)] but shall be capable of representing the values of all the members of the enumeration. The enumerated type is incomplete until immediately after the **`}`** that terminates the list of enumerator declarations, and complete thereafter.

5 EXAMPLE The following fragment:

```
enum hue { chartreuse, burgundy, claret=20, winedark };
enum hue col, *cp;
col = claret;
cp = &col;
if (*cp != burgundy)
      /* ... */
```

makes **`hue`** the tag of an enumeration, and then declares **`col`** as an object that has that type and **`cp`** as a pointer to an object that has that type. The enumerated values are in the set { 0, 1, 20, 21 }.

Forward references: tags (6.7.2.3).

6.7.2.3 Tags

Constraints

1 A specific type shall have its content defined at most once.

2 Where two declarations that use the same tag declare the same type, they shall both use the same choice of **`struct`**, **`union`**, or **`enum`**.

3 A type specifier of the form

`enum` *identifier*

without an enumerator list shall only appear after the type it specifies is complete.

---

127) Thus, the identifiers of enumeration constants declared in the same scope shall all be distinct from each other and from other identifiers declared in ordinary declarators.

128) An implementation may delay the choice of which integer type until all enumeration constants have been seen.

Semantics

4 All declarations of structure, union, or enumerated types that have the same scope and use the same tag declare the same type. Irrespective of whether there is a tag or what other declarations of the type are in the same translation unit, the type is incomplete[129] until immediately after the closing brace of the list defining the content, and complete thereafter.

5 Two declarations of structure, union, or enumerated types which are in different scopes or use different tags declare distinct types. Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type.

6 A type specifier of the form

> *struct-or-union identifier*$_{opt}$ `{` *struct-declaration-list* `}`

or

> **`enum`** *identifier*$_{opt}$ `{` *enumerator-list* `}`

or

> **`enum`** *identifier*$_{opt}$ `{` *enumerator-list* `,` `}`

declares a structure, union, or enumerated type. The list defines the *structure content*, *union content*, or *enumeration content*. If an identifier is provided,[130] the type specifier also declares the identifier to be the tag of that type.

7 A declaration of the form

> *struct-or-union identifier* `;`

specifies a structure or union type and declares the identifier as a tag of that type.[131]

8 If a type specifier of the form

> *struct-or-union identifier*

occurs other than as part of one of the above forms, and no other declaration of the identifier as a tag is visible, then it declares an incomplete structure or union type, and declares the identifier as the tag of that type.[131]

---

129) An incomplete type may only by used when the size of an object of that type is not needed. It is not needed, for example, when a typedef name is declared to be a specifier for a structure or union, or when a pointer to or a function returning a structure or union is being declared. (See incomplete types in 6.2.5.) The specification has to be complete before such a function is called or defined.

130) If there is no identifier, the type can, within the translation unit, only be referred to by the declaration of which it is a part. Of course, when the declaration is of a typedef name, subsequent declarations can make use of that typedef name to declare objects having the specified structure, union, or enumerated type.

131) A similar construction with **`enum`** does not exist.

9 If a type specifier of the form

struct-or-union identifier

or

enum *identifier*

occurs other than as part of one of the above forms, and a declaration of the identifier as a tag is visible, then it specifies the same type as that other declaration, and does not redeclare the tag.

10 EXAMPLE 1 This mechanism allows declaration of a self-referential structure.

```
struct tnode {
      int count;
      struct tnode *left, *right;
};
```

specifies a structure that contains an integer and two pointers to objects of the same type. Once this declaration has been given, the declaration

```
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**.

11 The following alternative formulation uses the **typedef** mechanism:

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

12 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;
```

may 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.

6.7.3 Type qualifiers

Syntax

1 *type-qualifier:*
`const`
`restrict`
`volatile`
`_Atomic`

Constraints

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

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

Semantics

4 The properties associated with qualified types are meaningful only for expressions that are lvalues.[132)]

5 If the same qualifier appears more than once in the same *specifier-qualifier-list*, either directly or via one or more **`typedef`**s, the behavior is the same as if it appeared only once. If other qualifiers appear along with the **`_Atomic`** qualifier in a *specifier-qualifier-*

132) The implementation may place a **`const`** object that is not **`volatile`** in a read-only region of storage. Moreover, the implementation need not allocate storage for such an object if its address is never used.

list, the resulting type is the so-qualified atomic type.

6 If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.[133)]

7 An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3. Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously.[134)] What constitutes an access to an object that has volatile-qualified type is implementation-defined.

8 An object that is accessed through a restrict-qualified pointer has a special association with that pointer. This association, defined in 6.7.3.1 below, requires that all accesses to that object use, directly or indirectly, the value of that particular pointer.[135)] The intended use of the **`restrict`** qualifier (like the **`register`** storage class) is to promote optimization, and deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior).

9 If the specification of an array type includes any type qualifiers, the element type is so-qualified, not the array type. If the specification of a function type includes any type qualifiers, the behavior is undefined.[136)]

10 For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.

11 EXAMPLE 1 An object declared

```
extern const volatile int real_time_clock;
```

---

133) This applies to those objects that behave as if they were defined with qualified types, even if they are never actually defined as objects in the program (such as an object at a memory-mapped input/output address).

134) A **`volatile`** declaration may be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function. Actions on objects so declared shall not be ''optimized out'' by an implementation or reordered except as permitted by the rules for evaluating expressions.

135) For example, a statement that assigns a value returned by **`malloc`** to a single pointer establishes this association between the allocated object and the pointer.

136) Both of these can occur through the use of **`typedef`**s.

may be modifiable by hardware, but cannot be assigned to, incremented, or decremented.

12 EXAMPLE 2 The following declarations and expressions illustrate the behavior when type qualifiers modify an aggregate type:

```
const struct s { int mem; } cs = { 1 };
struct s ncs;  // the object ncs is modifiable
typedef int A[2][3];
const A a = {{4, 5, 6}, {7, 8, 9}}; // array of array of const int
int *pi;
const int *pci;

ncs = cs;       // valid
cs = ncs;       // violates modifiable lvalue constraint for =
pi = &ncs.mem;  // valid
pi = &cs.mem;   // violates type constraints for =
pci = &cs.mem;  // valid
pi = a[0];      // invalid: a[0] has type "const int *"
```

13 EXAMPLE 3 The declaration

```
_Atomic volatile int *p;
```

specifies that **p** has the type ‘‘pointer to volatile atomic int’’, a pointer to a volatile-qualified atomic type.

6.7.3.1 Formal definition of `restrict`

1 Let **D** be a declaration of an ordinary identifier that provides a means of designating an object **P** as a restrict-qualified pointer to type **T**.

2 If **D** appears inside a block and does not have storage class **extern**, let **B** denote the block. If **D** appears in the list of parameter declarations of a function definition, let **B** denote the associated block. Otherwise, let **B** denote the block of **main** (or the block of whatever function is called at program startup in a freestanding environment).

3 In what follows, a pointer expression **E** is said to be *based* on object **P** if (at some sequence point in the execution of **B** prior to the evaluation of **E**) modifying **P** to point to a copy of the array object into which it formerly pointed would change the value of **E**.[137] Note that ‘‘based’’ is defined only for expressions with pointer types.

4 During each execution of **B**, let **L** be any lvalue that has **&L** based on **P**. If **L** is used to access the value of the object **X** that it designates, and **X** is also modified (by any means), then the following requirements apply: **T** shall not be const-qualified. Every other lvalue used to access the value of **X** shall also have its address based on **P**. Every access that modifies **X** shall be considered also to modify **P**, for the purposes of this subclause. If **P** is assigned the value of a pointer expression **E** that is based on another restricted pointer

137) In other words, **E** depends on the value of **P** itself rather than on the value of an object referenced indirectly through **P**. For example, if identifier **p** has type **(int **restrict)**, then the pointer expressions **p** and **p+1** are based on the restricted pointer object designated by **p**, but the pointer expressions ***p** and **p[1]** are not.

object **P2**, associated with block **B2**, then either the execution of **B2** shall begin before the execution of **B**, or the execution of **B2** shall end prior to the assignment. If these requirements are not met, then the behavior is undefined.

5 Here an execution of **B** means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration associated with **B**.

6 A translator is free to ignore any or all aliasing implications of uses of **restrict**.

7 EXAMPLE 1 The file scope declarations

```
int * restrict a;
int * restrict b;
extern int c[];
```

assert that if an object is accessed using one of **a**, **b**, or **c**, and that object is modified anywhere in the program, then it is never accessed using either of the other two.

8 EXAMPLE 2 The function parameter declarations in the following example

```
void f(int n, int * restrict p, int * restrict q)
{
      while (n-- > 0)
            *p++ = *q++;
}
```

assert that, during each execution of the function, if an object is accessed through one of the pointer parameters, then it is not also accessed through the other.

9 The benefit of the **restrict** qualifiers is that they enable a translator to make an effective dependence analysis of function **f** without examining any of the calls of **f** in the program. The cost is that the programmer has to examine all of those calls to ensure that none give undefined behavior. For example, the second call of **f** in **g** has undefined behavior because each of **d[1]** through **d[49]** is accessed through both **p** and **q**.

```
void g(void)
{
      extern int d[100];
      f(50, d + 50, d); // valid
      f(50, d +  1, d); // undefined behavior
}
```

10 EXAMPLE 3 The function parameter declarations

```
void h(int n, int * restrict p, int * restrict q, int * restrict r)
{
      int i;
      for (i = 0; i < n; i++)
            p[i] = q[i] + r[i];
}
```

illustrate how an unmodified object can be aliased through two restricted pointers. In particular, if **a** and **b** are disjoint arrays, a call of the form **h(100, a, b, b)** has defined behavior, because array **b** is not modified within function **h**.

11 EXAMPLE 4 The rule limiting assignments between restricted pointers does not distinguish between a function call and an equivalent nested block. With one exception, only ''outer-to-inner'' assignments between restricted pointers declared in nested blocks have defined behavior.

```
{
        int * restrict p1;
        int * restrict q1;
        p1 = q1; // undefined behavior
        {
                int * restrict p2 = p1; // valid
                int * restrict q2 = q1; // valid
                p1 = q2;                // undefined behavior
                p2 = q2;                // undefined behavior
        }
}
```

12 The one exception allows the value of a restricted pointer to be carried out of the block in which it (or, more precisely, the ordinary identifier used to designate it) is declared when that block finishes execution. For example, this permits **new_vector** to return a **vector**.

```
typedef struct { int n; float * restrict v; } vector;
vector new_vector(int n)
{
        vector t;
        t.n = n;
        t.v = malloc(n * sizeof (float));
        return t;
}
```

6.7.4 Function specifiers

Syntax

1 *function-specifier:*
  inline
  _Noreturn

Constraints

2 Function specifiers shall be used only in the declaration of an identifier for a function.

3 An inline definition of a function with external linkage shall not contain a definition of a modifiable object with static or thread storage duration, and shall not contain a reference to an identifier with internal linkage.

4 In a hosted environment, no function specifier(s) shall appear in a declaration of **main**.

Semantics

5 A function specifier may appear more than once; the behavior is the same as if it appeared only once.

6 A function declared with an **inline** function specifier is an *inline function*. Making a function an inline function suggests that calls to the function be as fast as possible.[138)]

The extent to which such suggestions are effective is implementation-defined.[139)]

7 Any function with internal linkage can be an inline function. For a function with external linkage, the following restrictions apply: If a function is declared with an **`inline`** function specifier, then it shall also be defined in the same translation unit. If all of the file scope declarations for a function in a translation unit include the **`inline`** function specifier without **`extern`**, then the definition in that translation unit is an *inline definition*. An inline definition does not provide an external definition for the function, and does not forbid an external definition in another translation unit. An inline definition provides an alternative to an external definition, which a translator may use to implement any call to the function in the same translation unit. It is unspecified whether a call to the function uses the inline definition or the external definition.[140)]

8 A function declared with a **`_Noreturn`** function specifier shall not return to its caller.

Recommended practice

9 The implementation should produce a diagnostic message for a function declared with a **`_Noreturn`** function specifier that appears to be capable of returning to its caller.

10 EXAMPLE 1 The declaration of an inline function with external linkage can result in either an external definition, or a definition available for use only within the translation unit. A file scope declaration with **`extern`** creates an external definition. The following example shows an entire translation unit.

```
inline double fahr(double t)
{
      return (9.0 * t) / 5.0 + 32.0;
}
inline double cels(double t)
{
      return (5.0 * (t - 32.0)) / 9.0;
}
extern double fahr(double);    // creates an external definition
```

---

138) By using, for example, an alternative to the usual function call mechanism, such as ‘‘inline substitution’’. Inline substitution is not textual substitution, nor does it create a new function. Therefore, for example, the expansion of a macro used within the body of the function uses the definition it had at the point the function body appears, and not where the function is called; and identifiers refer to the declarations in scope where the body occurs. Likewise, the function has a single address, regardless of the number of inline definitions that occur in addition to the external definition.

139) For example, an implementation might never perform inline substitution, or might only perform inline substitutions to calls in the scope of an **`inline`** declaration.

140) Since an inline definition is distinct from the corresponding external definition and from any other corresponding inline definitions in other translation units, all corresponding objects with static storage duration are also distinct in each of the definitions.

```
double convert(int is_fahr, double temp)
{
      /* A translator may perform inline substitutions */
      return is_fahr ? cels(temp) : fahr(temp);
}
```

11 Note that the definition of **fahr** is an external definition because **fahr** is also declared with **extern**, but the definition of **cels** is an inline definition. Because **cels** has external linkage and is referenced, an external definition has to appear in another translation unit (see 6.9); the inline definition and the external definition are distinct and either may be used for the call.

12 EXAMPLE 2

```
_Noreturn void f () {
      abort(); // ok
}

_Noreturn void g (int i) { // causes undefined behavior if i <= 0
      if (i > 0) abort();
}
```

Forward references: function definitions (6.9.1).

6.7.5 Alignment specifier

Syntax

1 *alignment-specifier:*
  _Alignas (*type-name* **)**
  _Alignas (*constant-expression* **)**

Constraints

2 An alignment attribute shall not be specified in a declaration of a typedef, or a bit-field, or a function, or a parameter, or an object declared with the **register** storage-class specifier.

3 The constant expression shall be an integer constant expression. It shall evaluate to a valid fundamental alignment, or to a valid extended alignment supported by the implementation in the context in which it appears, or to zero.

4 The combined effect of all alignment attributes in a declaration shall not specify an alignment that is less strict than the alignment that would otherwise be required for the type of the object or member being declared.

Semantics

5 The first form is equivalent to **_Alignas (_Alignof (***type-name***))**.

6 The alignment requirement of the declared object or member is taken to be the specified alignment. An alignment specification of zero has no effect.[141] When multiple alignment specifiers occur in a declaration, the effective alignment requirement is the strictest specified alignment.

7 If the definition of an object has an alignment specifier, any other declaration of that object shall either specify equivalent alignment or have no alignment specifier. If the definition of an object does not have an alignment specifier, any other declaration of that object shall also have no alignment specifier. If declarations of an object in different translation units have different alignment specifiers, the behavior is undefined.

6.7.6 Declarators

Syntax

1 *declarator:*
pointer$_{opt}$ *direct-declarator*

direct-declarator:
identifier
`(` *declarator* `)`
direct-declarator `[` *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ `]`
direct-declarator `[` **`static`** *type-qualifier-list*$_{opt}$ *assignment-expression* `]`
direct-declarator `[` *type-qualifier-list* **`static`** *assignment-expression* `]`
direct-declarator `[` *type-qualifier-list*$_{opt}$ `*` `]`
direct-declarator `(` *parameter-type-list* `)`
direct-declarator `(` *identifier-list*$_{opt}$ `)`

pointer:
`*` *type-qualifier-list*$_{opt}$
`*` *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
type-qualifier
type-qualifier-list *type-qualifier*

parameter-type-list:
parameter-list
parameter-list `, ...`

parameter-list:
parameter-declaration
parameter-list `,` *parameter-declaration*

parameter-declaration:
declaration-specifiers *declarator*
declaration-specifiers *abstract-declarator*$_{opt}$

141) An alignment specification of zero also does not affect other alignment specifications in the same declaration.

identifier-list:
identifier
identifier-list `,` *identifier*

Semantics

2 Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers.

3 A *full declarator* is a declarator that is not part of another declarator. The end of a full declarator is a sequence point. If, in the nested sequence of declarators in a full declarator, there is a declarator specifying a variable length array type, the type specified by the full declarator is said to be *variably modified*. Furthermore, any type derived by declarator type derivation from a variably modified type is itself variably modified.

4 In the following subclauses, consider a declaration

```
T D1
```

where **`T`** contains the declaration specifiers that specify a type *T* (such as **`int`**) and **`D1`** is a declarator that contains an identifier *ident*. The type specified for the identifier *ident* in the various forms of declarator is described inductively using this notation.

5 If, in the declaration ‘‘**`T D1`**’’, **`D1`** has the form

identifier

then the type specified for *ident* is *T*.

6 If, in the declaration ‘‘**`T D1`**’’, **`D1`** has the form

```
( D )
```

then *ident* has the type specified by the declaration ‘‘**`T D`**’’. Thus, a declarator in parentheses is identical to the unparenthesized declarator, but the binding of complicated declarators may be altered by parentheses.

Implementation limits

7 As discussed in 5.2.4.1, an implementation may limit the number of pointer, array, and function declarators that modify an arithmetic, structure, union, or **`void`** type, either directly or via one or more **`typedef`**s.

Forward references: array declarators (6.7.6.2), type definitions (6.7.8).

6.7.6.1 Pointer declarators

Semantics

1 If, in the declaration ‘‘`T D1`’’, `D1` has the form

`*` *type-qualifier-list*$_{opt}$ `D`

and the type specified for *ident* in the declaration ‘‘`T D`’’ is ‘‘*derived-declarator-type-list T*’’, then the type specified for *ident* is ‘‘*derived-declarator-type-list type-qualifier-list* pointer to *T*’’. For each type qualifier in the list, *ident* is a so-qualified pointer.

2 For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types.

3 EXAMPLE The following pair of declarations demonstrates the difference between a ‘‘variable pointer to a constant value’’ and a ‘‘constant pointer to a variable value’’.

```
const int *ptr_to_constant;
int *const constant_ptr;
```

The contents of any object pointed to by `ptr_to_constant` shall not be modified through that pointer, but `ptr_to_constant` itself may be changed to point to another object. Similarly, the contents of the `int` pointed to by `constant_ptr` may be modified, but `constant_ptr` itself shall always point to the same location.

4 The declaration of the constant pointer `constant_ptr` may be clarified by including a definition for the type ‘‘pointer to `int`’’.

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

declares `constant_ptr` as an object that has type ‘‘const-qualified pointer to `int`’’.

6.7.6.2 Array declarators

Constraints

1 In addition to optional type qualifiers and the keyword `static`, the `[` and `]` may delimit an expression or `*`. If they delimit an expression (which specifies the size of an array), the expression shall have an integer type. If the expression is a constant expression, it shall have a value greater than zero. The element type shall not be an incomplete or function type. The optional type qualifiers and the keyword `static` shall appear only in a declaration of a function parameter with an array type, and then only in the outermost array type derivation.

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

Semantics

3 If, in the declaration ``**T D1**'', **D1** has one of the forms:

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

and the type specified for *ident* in the declaration ``**T D**'' is ``*derived-declarator-type-list T*'', then the type specified for *ident* is ``*derived-declarator-type-list* array of *T*''.[142] (See 6.7.6.3 for the meaning of the optional type qualifiers and the keyword **static**.)

4 If the size is not present, the array type is an incomplete type. If the size is ***** instead of being an expression, the array type is a *variable length array* type of unspecified size, which can only be used in declarations or type names with function prototype scope;[143] such arrays are nonetheless complete types. If the size is an integer constant expression and the element type has a known constant size, the array type is not a variable length array type; otherwise, the array type is a *variable length array* type. (Variable length arrays are a conditional feature that implementations need not support; see 6.10.8.3.)

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

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

7 EXAMPLE 1

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

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

8 EXAMPLE 2 Note the distinction between the declarations

---

142) When several ``array of'' specifications are adjacent, a multidimensional array is declared.

143) Thus, ***** can be used only in function declarations that are not definitions (see 6.7.6.3).

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

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

9 EXAMPLE 3 The following declarations demonstrate the compatibility rules for variably modified types.

```
extern int n;
extern int m;

void fcompat(void)
{
      int a[n][6][m];
      int (*p)[4][n+1];
      int c[n][n][6][m];
      int (*r)[n][n][n+1];
      p = a;       // invalid: not compatible because 4 != 6
      r = c;       // compatible, but defined behavior only if
                   // n == 6 and m == n+1
}
```

10 EXAMPLE 4 All declarations of variably modified (VM) types have to be at either block scope or function prototype scope. Array objects declared with the **_Thread_local**, **static**, or **extern** storage-class specifier cannot have a variable length array (VLA) type. However, an object declared with the **static** storage-class specifier can have a VM type (that is, a pointer to a VLA type). Finally, all identifiers declared with a VM type have to be ordinary identifiers and cannot, therefore, be members of structures or unions.

```
extern int n;
int A[n];                        // invalid: file scope VLA
extern int (*p2)[n];             // invalid: file scope VM
int B[100];                      // valid: file scope but not VM

void fvla(int m, int C[m][m]);   // valid: VLA with prototype scope

void fvla(int m, int C[m][m])    // valid: adjusted to auto pointer to VLA
{
      typedef int VLA[m][m];     // valid: block scope typedef VLA

      struct tag {
            int (*y)[n];         // invalid: y not ordinary identifier
            int z[n];            // invalid: z not ordinary identifier
      };
      int D[m];                  // valid: auto VLA
      static int E[m];           // invalid: static block scope VLA
      extern int F[m];           // invalid: F has linkage and is VLA
      int (*s)[m];               // valid: auto pointer to VLA
      extern int (*r)[m];        // invalid: r has linkage and points to VLA
      static int (*q)[m] = &B;   // valid: q is a static block pointer to VLA
}
```

Forward references: function declarators (6.7.6.3), function definitions (6.9.1), initialization (6.7.9).

6.7.6.3 Function declarators (including prototypes)

Constraints

1 A function declarator shall not specify a return type that is a function type or an array type.

2 The only storage-class specifier that shall occur in a parameter declaration is **`register`**.

3 An identifier list in a function declarator that is not part of a definition of that function shall be empty.

4 After adjustment, the parameters in a parameter type list in a function declarator that is part of a definition of that function shall not have incomplete type.

Semantics

5 If, in the declaration ‘‘`T D1`’’, `D1` has the form

`D (` *parameter-type-list* `)`

or

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

and the type specified for *ident* in the declaration ‘‘`T D`’’ is ‘‘*derived-declarator-type-list T*’’, then the type specified for *ident* is ‘‘*derived-declarator-type-list* function returning *T*’’.

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

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

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

9 If the list terminates with an ellipsis (`, ...`), no information about the number or types of the parameters after the comma is supplied.[144)]

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

---

144) The macros defined in the **`<stdarg.h>`** header (7.16) may be used to access arguments that correspond to the ellipsis.

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

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

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

14 An identifier list declares only the identifiers of the parameters of the function. An empty list in a function declarator that is part of a definition of that function specifies that the function has no parameters. The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.[145)]

15 For two function types to be compatible, both shall specify compatible return types.[146)] Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If one type has a parameter type list and the other type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list, the parameter list shall not have an ellipsis terminator and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions. If one type has a parameter type list and the other type is specified by a function definition that contains a (possibly empty) identifier list, both shall agree in the number of parameters, and the type of each prototype parameter shall be compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier. (In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the adjusted type and each parameter declared with qualified type is taken as having the unqualified version of its declared type.)

16 EXAMPLE 1 The declaration

```
int f(void), *fip(), (*pfi)();
```

declares a function **f** with no parameters returning an **int**, a function **fip** with no parameter specification returning a pointer to an **int**, and a pointer **pfi** to a function with no parameter specification returning an **int**. It is especially useful to compare the last two. The binding of ***fip()** is ***(fip())**, so that the declaration suggests, and the same construction in an expression requires, the calling of a function **fip**, and then using indirection through the pointer result to yield an **int**. In the declarator **(*pfi)()**, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function

145) See ‘‘future language directions’’ (6.11.6).

146) If both function types are ‘‘old style’’, parameter types are not compared.

designator, which is then used to call the function; it returns an **int**.

17 If the declaration occurs outside of any function, the identifiers have file scope and external linkage. If the declaration occurs inside a function, the identifiers of the functions **f** and **fip** have block scope and either internal or external linkage (depending on what file scope declarations for these identifiers are visible), and the identifier of the pointer **pfi** has block scope and no linkage.

18 EXAMPLE 2 The declaration

```
int (*apfi[3])(int *x, int *y);
```

declares an array **apfi** of three pointers to functions returning **int**. Each of these functions has two parameters that are pointers to **int**. The identifiers **x** and **y** are declared for descriptive purposes only and go out of scope at the end of the declaration of **apfi**.

19 EXAMPLE 3 The declaration

```
int (*fpfi(int (*)(long), int))(int, ...);
```

declares a function **fpfi** that returns a pointer to a function returning an **int**. The function **fpfi** has two parameters: a pointer to a function returning an **int** (with one parameter of type **long int**), and an **int**. The pointer returned by **fpfi** points to a function that has one **int** parameter and accepts zero or more additional arguments of any type.

20 EXAMPLE 4 The following prototype has a variably modified parameter.

```
void addscalar(int n, int m,
      double a[n][n*m+300], double x);

int main()
{
      double b[4][308];
      addscalar(4, 2, b, 2.17);
      return 0;
}

void addscalar(int n, int m,
      double a[n][n*m+300], double x)
{
      for (int i = 0; i < n; i++)
            for (int j = 0, k = n*m+300; j < k; j++)
                  // a is a pointer to a VLA with n*m+300 elements
                  a[i][j] += x;
}
```

21 EXAMPLE 5 The following are all compatible function prototype declarators.

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

as are:

```
void f(double (* restrict a)[5]);
void f(double a[restrict][5]);
void f(double a[restrict 3][5]);
void f(double a[restrict static 3][5]);
```

(Note that the last declaration also specifies that the argument corresponding to **a** in any call to **f** must be a non-null pointer to the first of at least three arrays of 5 doubles, which the others do not.)

Forward references: function definitions (6.9.1), type names (6.7.7).

6.7.7 Type names

Syntax

1 *type-name:*
  specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator:
  pointer
  pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:
  (*abstract-declarator* **)**
  direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list*$_{opt}$
    assignment-expression$_{opt}$ **]**
  direct-abstract-declarator$_{opt}$ **[static** *type-qualifier-list*$_{opt}$
    assignment-expression **]**
  direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list* **static**
    assignment-expression **]**
  direct-abstract-declarator$_{opt}$ **[*]**
  direct-abstract-declarator$_{opt}$ **(** *parameter-type-list*$_{opt}$ **)**

Semantics

2 In several contexts, it is necessary to specify a type. This is accomplished using a *type name*, which is syntactically a declaration for a function or an object of that type that omits the identifier.[147)]

3 EXAMPLE The constructions

```
(a) int
(b) int *
(c) int *[3]
(d) int (*)[3]
(e) int (*)[*]
(f) int *()
(g) int (*)(void)
(h) int (*const [])(unsigned int, ...)
```

name respectively the types (a) **int**, (b) pointer to **int**, (c) array of three pointers to **int**, (d) pointer to an array of three **int**s, (e) pointer to a variable length array of an unspecified number of **int**s, (f) function with no parameter specification returning a pointer to **int**, (g) pointer to function with no parameters

147) As indicated by the syntax, empty parentheses in a type name are interpreted as ‘‘function with no parameter specification’’, rather than redundant parentheses around the omitted identifier.

returning an **`int`**, and (h) array of an unspecified number of constant pointers to functions, each with one parameter that has type **`unsigned int`** and an unspecified number of other parameters, returning an **`int`**.

6.7.8 Type definitions

Syntax

1 *typedef-name:*
identifier

Constraints

2 If a typedef name specifies a variably modified type then it shall have block scope.

Semantics

3 In a declaration whose storage-class specifier is **`typedef`**, each declarator defines an identifier to be a typedef name that denotes the type specified for the identifier in the way described in 6.7.6. Any array size expressions associated with variable length array declarators are evaluated each time the declaration of the typedef name is reached in the order of execution. A **`typedef`** declaration does not introduce a new type, only a synonym for the type so specified. That is, in the following declarations:

```
typedef T type_ident;
type_ident D;
```

`type_ident` is defined as a typedef name with the type specified by the declaration specifiers in **`T`** (known as *T*), and the identifier in **`D`** has the type ‘‘*derived-declarator-type-list T*’’ where the *derived-declarator-type-list* is specified by the declarators of **`D`**. A typedef name shares the same name space as other identifiers declared in ordinary declarators.

4 EXAMPLE 1 After

```
typedef int MILES, KLICKSP();
typedef struct { double hi, lo; } range;
```

the constructions

```
MILES distance;
extern KLICKSP *metricp;
range x;
range z, *zp;
```

are all valid declarations. The type of **`distance`** is **`int`**, that of **`metricp`** is ‘‘pointer to function with no parameter specification returning **`int`**’’, and that of **`x`** and **`z`** is the specified structure; **`zp`** is a pointer to such a structure. The object **`distance`** has a type compatible with any other **`int`** object.

5 EXAMPLE 2 After the declarations

```
typedef struct s1 { int x; } t1, *tp1;
typedef struct s2 { int x; } t2, *tp2;
```

type **`t1`** and the type pointed to by **`tp1`** are compatible. Type **`t1`** is also compatible with type **`struct`**

s1, but not compatible with the types **struct s2**, **t2**, the type pointed to by **tp2**, or **int**.

6 EXAMPLE 3 The following obscure constructions

```
typedef signed int t;
typedef int plain;
struct tag {
      unsigned t:4;
      const t:5;
      plain r:5;
};
```

declare a typedef name **t** with type **signed int**, a typedef name **plain** with type **int**, and a structure with three bit-field members, one named **t** that contains values in the range [0, 15], an unnamed const-qualified bit-field which (if it could be accessed) would contain values in either the range [−15, +15] or [−16, +15], and one named **r** that contains values in one of the ranges [0, 31], [−15, +15], or [−16, +15]. (The choice of range is implementation-defined.) The first two bit-field declarations differ in that **unsigned** is a type specifier (which forces **t** to be the name of a structure member), while **const** is a type qualifier (which modifies **t** which is still visible as a typedef name). If these declarations are followed in an inner scope by

```
t f(t (t));
long t;
```

then a function **f** is declared with type ‘‘function returning **signed int** with one unnamed parameter with type pointer to function returning **signed int** with one unnamed parameter with type **signed int**’’, and an identifier **t** with type **long int**.

7 EXAMPLE 4 On the other hand, typedef names can be used to improve code readability. All three of the following declarations of the **signal** function specify exactly the same type, the first without making use of any typedef names.

```
typedef void fv(int), (*pfv)(int);

void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

8 EXAMPLE 5 If a typedef name denotes a variable length array type, the length of the array is fixed at the time the typedef name is defined, not each time it is used:

```
void copyt(int n)
{
      typedef int B[n];    // B is n ints, n evaluated now
      n += 1;
      B a;                 // a is n ints, n without += 1
      int b[n];            // a and b are different sizes
      for (int i = 1; i < n; i++)
            a[i-1] = b[i];
}
```

6.7.9 Initialization

Syntax

1 *initializer:*
  assignment-expression
  `{` *initializer-list* `}`
  `{` *initializer-list* `,` `}`

initializer-list:
  designation$_{opt}$ *initializer*
  initializer-list `,` *designation*$_{opt}$ *initializer*

designation:
  designator-list `=`

designator-list:
  designator
  designator-list designator

designator:
  `[` *constant-expression* `]`
  `.` *identifier*

Constraints

2 No initializer shall attempt to provide a value for an object not contained within the entity being initialized.

3 The type of the entity to be initialized shall be an array of unknown size or a complete object type that is not a variable length array type.

4 All the expressions in an initializer for an object that has static or thread storage duration shall be constant expressions or string literals.

5 If the declaration of an identifier has block scope, and the identifier has external or internal linkage, the declaration shall have no initializer for the identifier.

6 If a designator has the form

`[` *constant-expression* `]`

then the current object (defined below) shall have array type and the expression shall be an integer constant expression. If the array is of unknown size, any nonnegative value is valid.

7 If a designator has the form

`.` *identifier*

then the current object (defined below) shall have structure or union type and the identifier shall be the name of a member of that type.

Semantics

8 An initializer specifies the initial value stored in an object.

9 Except where explicitly stated otherwise, for the purposes of this subclause unnamed members of objects of structure and union type do not participate in initialization. Unnamed members of structure objects have indeterminate value even after initialization.

10 If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate. If an object that has static or thread storage duration is not initialized explicitly, then:

— if it has pointer type, it is initialized to a null pointer;

— if it has arithmetic type, it is initialized to (positive or unsigned) zero;

— if it is an aggregate, every member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;

— if it is a union, the first named member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;

11 The initializer for a scalar shall be a single expression, optionally enclosed in braces. The initial value of the object is that of the expression (after conversion); the same type constraints and conversions as for simple assignment apply, taking the type of the scalar to be the unqualified version of its declared type.

12 The rest of this subclause deals with initializers for objects that have aggregate or union type.

13 The initializer for a structure or union object that has automatic storage duration shall be either an initializer list as described below, or a single expression that has compatible structure or union type. In the latter case, the initial value of the object, including unnamed members, is that of the expression.

14 An array of character type may be initialized by a character string literal or UTF−8 string literal, optionally enclosed in braces. Successive bytes of the string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array.

15 An array with element type compatible with a qualified or unqualified version of **`wchar_t`**, **`char16_t`**, or **`char32_t`** may be initialized by a wide string literal with the corresponding encoding prefix (**`L`**, **`u`**, or **`U`**, respectively), optionally enclosed in braces. Successive wide characters of the wide string literal (including the terminating null wide character if there is room or if the array is of unknown size) initialize the elements of the array.

16 Otherwise, the initializer for an object that has aggregate or union type shall be a brace-enclosed list of initializers for the elements or named members.

17 Each brace-enclosed initializer list has an associated *current object*. When no designations are present, subobjects of the current object are initialized in order according to the type of the current object: array elements in increasing subscript order, structure members in declaration order, and the first named member of a union.[148] In contrast, a designation causes the following initializer to begin initialization of the subobject described by the designator. Initialization then continues forward in order, beginning with the next subobject after that described by the designator.[149]

18 Each designator list begins its description with the current object associated with the closest surrounding brace pair. Each item in the designator list (in order) specifies a particular member of its current object and changes the current object for the next designator (if any) to be that member.[150] The current object that results at the end of the designator list is the subobject to be initialized by the following initializer.

19 The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject;[151] all subobjects that are not initialized explicitly shall be initialized implicitly the same as objects that have static storage duration.

20 If the aggregate or union contains elements or members that are aggregates or unions, these rules apply recursively to the subaggregates or contained unions. If the initializer of a subaggregate or contained union begins with a left brace, the initializers enclosed by that brace and its matching right brace initialize the elements or members of the subaggregate or the contained union. Otherwise, only enough initializers from the list are taken to account for the elements or members of the subaggregate or the first member of the contained union; any remaining initializers are left to initialize the next element or member of the aggregate of which the current subaggregate or contained union is a part.

21 If there are fewer initializers in a brace-enclosed list than there are elements or members of an aggregate, or fewer characters in a string literal used to initialize an array of known size than there are elements in the array, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.

---

148) If the initializer list for a subaggregate or contained union does not begin with a left brace, its subobjects are initialized as usual, but the subaggregate or contained union does not become the current object: current objects are associated only with brace-enclosed initializer lists.

149) After a union member is initialized, the next object is not the next member of the union; instead, it is the next subobject of an object containing the union.

150) Thus, a designator can only specify a strict subobject of the aggregate or union that is associated with the surrounding brace pair. Note, too, that each separate designator list is independent.

151) Any initializer for the subobject which is overridden and so not used to initialize that subobject might not be evaluated at all.

22 If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. The array type is completed at the end of its initializer list.

23 The evaluations of the initialization list expressions are indeterminately sequenced with respect to one another and thus the order in which any side effects occur is unspecified.[152)]

24 EXAMPLE 1 Provided that `<complex.h>` has been `#include`d, the declarations

```
int i = 3.5;
double complex c = 5 + 3 * I;
```

define and initialize `i` with the value 3 and `c` with the value $5.0 + i3.0$.

25 EXAMPLE 2 The declaration

```
int x[] = { 1, 3, 5 };
```

defines and initializes `x` as a one-dimensional array object that has three elements, as no size was specified and there are three initializers.

26 EXAMPLE 3 The declaration

```
int y[4][3] = {
      { 1, 3, 5 },
      { 2, 4, 6 },
      { 3, 5, 7 },
};
```

is a definition with a fully bracketed initialization: 1, 3, and 5 initialize the first row of `y` (the array object `y[0]`), namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early, so `y[3]` is initialized with zeros. Precisely the same effect could have been achieved by

```
int y[4][3] = {
      1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y[0]` does not begin with a left brace, so three items from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`.

27 EXAMPLE 4 The declaration

```
int z[4][3] = {
      { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `z` as specified and initializes the rest with zeros.

28 EXAMPLE 5 The declaration

```
struct { int a[3], b; } w[] = { { 1 }, 2 };
```

is a definition with an inconsistently bracketed initialization. It defines an array with two element

152) In particular, the evaluation order need not be the same as the order of subobject initialization.

structures: **w[0].a[0]** is 1 and **w[1].a[0]** is 2; all the other elements are zero.

29 EXAMPLE 6 The declaration

```
short q[4][3][2] = {
      { 1 },
      { 2, 3 },
      { 4, 5, 6 }
};
```

contains an incompletely but consistently bracketed initialization. It defines a three-dimensional array object: **q[0][0][0]** is 1, **q[1][0][0]** is 2, **q[1][0][1]** is 3, and 4, 5, and 6 initialize **q[2][0][0]**, **q[2][0][1]**, and **q[2][1][0]**, respectively; all the rest are zero. The initializer for **q[0][0]** does not begin with a left brace, so up to six items from the current list may be used. There is only one, so the values for the remaining five elements are initialized with zero. Likewise, the initializers for **q[1][0]** and **q[2][0]** do not begin with a left brace, so each uses up to six items, initializing their respective two-dimensional subaggregates. If there had been more than six items in any of the lists, a diagnostic message would have been issued. The same initialization result could have been achieved by:

```
short q[4][3][2] = {
      1, 0, 0, 0, 0, 0,
      2, 3, 0, 0, 0, 0,
      4, 5, 6
};
```

or by:

```
short q[4][3][2] = {
      {
            { 1 },
      },
      {
            { 2, 3 },
      },
      {
            { 4, 5 },
            { 6 },
      }
};
```

in a fully bracketed form.

30 Note that the fully bracketed and minimally bracketed forms of initialization are, in general, less likely to cause confusion.

31 EXAMPLE 7 One form of initialization that completes array types involves typedef names. Given the declaration

```
typedef int A[];    // OK - declared with block scope
```

the declaration

```
A a = { 1, 2 }, b = { 3, 4, 5 };
```

is identical to

```
int a[] = { 1, 2 }, b[] = { 3, 4, 5 };
```

due to the rules for incomplete types.

32 EXAMPLE 8 The declaration

```
char s[] = "abc", t[3] = "abc";
```

defines ‘‘plain’’ **char** array objects **s** and **t** whose elements are initialized with character string literals. This declaration is identical to

```
char s[] = { 'a', 'b', 'c', '\0' },
     t[] = { 'a', 'b', 'c' };
```

The contents of the arrays are modifiable. On the other hand, the declaration

```
char *p = "abc";
```

defines **p** with type ‘‘pointer to **char**’’ and initializes it to point to an object with type ‘‘array of **char**’’ with length 4 whose elements are initialized with a character string literal. If an attempt is made to use **p** to modify the contents of the array, the behavior is undefined.

33 EXAMPLE 9 Arrays can be initialized to correspond to the elements of an enumeration by using designators:

```
enum { member_one, member_two };
const char *nm[] = {
      [member_two] = "member two",
      [member_one] = "member one",
};
```

34 EXAMPLE 10 Structure members can be initialized to nonzero values without depending on their order:

```
div_t answer = { .quot = 2, .rem = -1 };
```

35 EXAMPLE 11 Designators can be used to provide explicit initialization when unadorned initializer lists might be misunderstood:

```
struct { int a[3], b; } w[] =
      { [0].a = {1}, [1].a[0] = 2 };
```

36 EXAMPLE 12 Space can be ‘‘allocated’’ from both ends of an array by using a single designator:

```
int a[MAX] = {
      1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

37 In the above, if **MAX** is greater than ten, there will be some zero-valued elements in the middle; if it is less than ten, some of the values provided by the first five initializers will be overridden by the second five.

38 EXAMPLE 13 Any member of a union can be initialized:

```
union { /* ... */ } u = { .any_member = 42 };
```

Forward references: common definitions **<stddef.h>** (7.19).

6.7.10 Static assertions

Syntax

1 *static_assert-declaration:*
`_Static_assert (` *constant-expression* `,` *string-literal* `) ;`

Constraints

2 The constant expression shall compare unequal to 0.

Semantics

3 The constant expression shall be an integer constant expression. If the value of the constant expression compares unequal to 0, the declaration has no effect. Otherwise, the constraint is violated and the implementation shall produce a diagnostic message that includes the text of the string literal, except that characters not in the basic source character set are not required to appear in the message.

Forward references: diagnostics (7.2).

6.8 Statements and blocks

Syntax

1 *statement:*
labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement

Semantics

2 A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence.

3 A *block* allows a set of declarations and statements to be grouped into one syntactic unit. The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear.

4 A *full expression* is an expression that is not part of another expression or of a declarator. Each of the following is a full expression: an initializer that is not part of a compound literal; the expression in an expression statement; the controlling expression of a selection statement (**`if`** or **`switch`**); the controlling expression of a **`while`** or **`do`** statement; each of the (optional) expressions of a **`for`** statement; the (optional) expression in a **`return`** statement. There is a sequence point between the evaluation of a full expression and the evaluation of the next full expression to be evaluated.

Forward references: expression and null statements (6.8.3), selection statements (6.8.4), iteration statements (6.8.5), the **`return`** statement (6.8.6.4).

6.8.1 Labeled statements

Syntax

1 *labeled-statement:*
identifier **`:`** *statement*
`case` *constant-expression* **`:`** *statement*
`default :` *statement*

Constraints

2 A **`case`** or **`default`** label shall appear only in a **`switch`** statement. Further constraints on such labels are discussed under the **`switch`** statement.

3 Label names shall be unique within a function.

Semantics

4 Any statement may be preceded by a prefix that declares an identifier as a label name. Labels in themselves do not alter the flow of control, which continues unimpeded across them.

Forward references: the `goto` statement (6.8.6.1), the `switch` statement (6.8.4.2).

6.8.2 Compound statement

Syntax

1 *compound-statement:*
  `{` *block-item-list*$_{opt}$ `}`

block-item-list:
  block-item
  block-item-list block-item

block-item:
  declaration
  statement

Semantics

2 A *compound statement* is a block.

6.8.3 Expression and null statements

Syntax

1 *expression-statement:*
  expression$_{opt}$ `;`

Semantics

2 The expression in an expression statement is evaluated as a void expression for its side effects.[153)]

3 A *null statement* (consisting of just a semicolon) performs no operations.

4 EXAMPLE 1 If a function call is evaluated as an expression statement for its side effects only, the discarding of its value may be made explicit by converting the expression to a void expression by means of a cast:

```
int p(int);
/* ... */
(void)p(0);
```

153) Such as assignments, and function calls which have side effects.

5 EXAMPLE 2 In the program fragment

```
char *s;
/* ... */
while (*s++ != '\0')
        ;
```

a null statement is used to supply an empty loop body to the iteration statement.

6 EXAMPLE 3 A null statement may also be used to carry a label just before the closing `}` of a compound statement.

```
while (loop1) {
        /* ... */
        while (loop2) {
                /* ... */
                if (want_out)
                        goto end_loop1;
                /* ... */
        }
        /* ... */
end_loop1: ;
}
```

Forward references: iteration statements (6.8.5).

6.8.4 Selection statements

Syntax

1 *selection-statement:*
  `if` **`(`** *expression* **`)`** *statement*
  `if` **`(`** *expression* **`)`** *statement* **`else`** *statement*
  `switch` **`(`** *expression* **`)`** *statement*

Semantics

2 A selection statement selects among a set of statements depending on the value of a controlling expression.

3 A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement.

6.8.4.1 The `if` statement

Constraints

1 The controlling expression of an **`if`** statement shall have scalar type.

Semantics

2 In both forms, the first substatement is executed if the expression compares unequal to 0. In the **`else`** form, the second substatement is executed if the expression compares equal

to 0. If the first substatement is reached via a label, the second substatement is not executed.

3 An **`else`** is associated with the lexically nearest preceding **`if`** that is allowed by the syntax.

6.8.4.2 The `switch` statement

Constraints

1 The controlling expression of a **`switch`** statement shall have integer type.

2 If a **`switch`** statement has an associated **`case`** or **`default`** label within the scope of an identifier with a variably modified type, the entire **`switch`** statement shall be within the scope of that identifier.[154)]

3 The expression of each **`case`** label shall be an integer constant expression and no two of the **`case`** constant expressions in the same **`switch`** statement shall have the same value after conversion. There may be at most one **`default`** label in a **`switch`** statement. (Any enclosed **`switch`** statement may have a **`default`** label or **`case`** constant expressions with values that duplicate **`case`** constant expressions in the enclosing **`switch`** statement.)

Semantics

4 A **`switch`** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression, and on the presence of a **`default`** label and the values of any **`case`** labels on or in the switch body. A **`case`** or **`default`** label is accessible only within the closest enclosing **`switch`** statement.

5 The integer promotions are performed on the controlling expression. The constant expression in each **`case`** label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched **`case`** label. Otherwise, if there is a **`default`** label, control jumps to the labeled statement. If no converted **`case`** constant expression matches and there is no **`default`** label, no part of the switch body is executed.

Implementation limits

6 As discussed in 5.2.4.1, the implementation may limit the number of **`case`** values in a **`switch`** statement.

154) That is, the declaration either precedes the **`switch`** statement, or it follows the last **`case`** or **`default`** label associated with the **`switch`** that is in the block containing the declaration.

7 EXAMPLE In the artificial program fragment

```
switch (expr)
{
      int i = 4;
      f(i);
case 0:
      i = 17;
      /* falls through into default code */
default:
      printf("%d\n", i);
}
```

the object whose identifier is **i** exists with automatic storage duration (within the block) but is never initialized, and thus if the controlling expression has a nonzero value, the call to the **printf** function will access an indeterminate value. Similarly, the call to the function **f** cannot be reached.

6.8.5 Iteration statements

Syntax

1 *iteration-statement:*
    while (*expression* **)** *statement*
    do *statement* **while (** *expression* **) ;**
    for ($expression_{opt}$ **;** $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*
    for (*declaration* $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*

Constraints

2 The controlling expression of an iteration statement shall have scalar type.

3 The declaration part of a **for** statement shall only declare identifiers for objects having storage class **auto** or **register**.

Semantics

4 An iteration statement causes a statement called the *loop body* to be executed repeatedly until the controlling expression compares equal to 0. The repetition occurs regardless of whether the loop body is entered from the iteration statement or by a jump.[155)]

5 An iteration statement is a block whose scope is a strict subset of the scope of its enclosing block. The loop body is also a block whose scope is a strict subset of the scope of the iteration statement.

6 An iteration statement whose controlling expression is not a constant expression,[156)] that performs no input/output operations, does not access volatile objects, and performs no synchronization or atomic operations in its body, controlling expression, or (in the case of

155) Code jumped over is not executed. In particular, the controlling expression of a **for** or **while** statement is not evaluated before entering the loop body, nor is *clause-1* of a **for** statement.

156) An omitted controlling expression is replaced by a nonzero constant, which is a constant expression.

a **`for`** statement) its *expression-3*, may be assumed by the implementation to terminate.[157)]

6.8.5.1 The `while` statement

1 The evaluation of the controlling expression takes place before each execution of the loop body.

6.8.5.2 The `do` statement

1 The evaluation of the controlling expression takes place after each execution of the loop body.

6.8.5.3 The `for` statement

1 The statement

`for (` *clause-1* **`;`** *expression-2* **`;`** *expression-3* **`)`** *statement*

behaves as follows: The expression *expression-2* is the controlling expression that is evaluated before each execution of the loop body. The expression *expression-3* is evaluated as a void expression after each execution of the loop body. If *clause-1* is a declaration, the scope of any identifiers it declares is the remainder of the declaration and the entire loop, including the other two expressions; it is reached in the order of execution before the first evaluation of the controlling expression. If *clause-1* is an expression, it is evaluated as a void expression before the first evaluation of the controlling expression.[158)]

2 Both *clause-1* and *expression-3* can be omitted. An omitted *expression-2* is replaced by a nonzero constant.

6.8.6 Jump statements

Syntax

1 *jump-statement:*
  `goto` *identifier* **`;`**
  `continue ;`
  `break ;`
  `return` *expression*$_{opt}$ **`;`**

---

157) This is intended to allow compiler transformations such as removal of empty loops even when termination cannot be proven.

158) Thus, *clause-1* specifies initialization for the loop, possibly declaring one or more variables for use in the loop; the controlling expression, *expression-2*, specifies an evaluation made before each iteration, such that execution of the loop continues until the expression compares equal to 0; and *expression-3* specifies an operation (such as incrementing) that is performed after each iteration.

Semantics

2 A jump statement causes an unconditional jump to another place.

6.8.6.1 The `goto` statement

Constraints

1 The identifier in a **`goto`** statement shall name a label located somewhere in the enclosing function. A **`goto`** statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier.

Semantics

2 A **`goto`** statement causes an unconditional jump to the statement prefixed by the named label in the enclosing function.

3 EXAMPLE 1 It is sometimes convenient to jump into the middle of a complicated set of statements. The following outline presents one possible approach to a problem based on these three assumptions:

1. The general initialization code accesses objects only visible to the current function.
2. The general initialization code is too large to warrant duplication.
3. The code to determine the next operation is at the head of the loop. (To allow it to be reached by **`continue`** statements, for example.)

```
/* ... */
goto first_time;
for (;;) {
        // determine next operation
        /* ... */
        if (need to reinitialize) {
                // reinitialize-only code
                /* ... */
        first_time:
                // general initialization code
                /* ... */
                continue;
        }
        // handle other operations
        /* ... */
}
```

4 EXAMPLE 2 A **goto** statement is not allowed to jump past any declarations of objects with variably modified types. A jump within the scope, however, is permitted.

```
goto lab3;                  // invalid: going INTO scope of VLA.
{
      double a[n];
      a[j] = 4.4;
lab3:
      a[j] = 3.3;
      goto lab4;            // valid: going WITHIN scope of VLA.
      a[j] = 5.5;
lab4:
      a[j] = 6.6;
}
goto lab4;                  // invalid: going INTO scope of VLA.
```

6.8.6.2 The `continue` statement

Constraints

1 A **continue** statement shall appear only in or as a loop body.

Semantics

2 A **continue** statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body. More precisely, in each of the statements

```
while (/* ... */) {      do {                      for (/* ... */) {
   /* ... */                /* ... */                  /* ... */
   continue;                continue;                  continue;
   /* ... */                /* ... */                  /* ... */
contin: ;                 contin: ;                 contin: ;
}                         } while (/* ... */);      }
```

unless the **continue** statement shown is in an enclosed iteration statement (in which case it is interpreted within that statement), it is equivalent to **goto contin;**.[159)]

6.8.6.3 The `break` statement

Constraints

1 A **break** statement shall appear only in or as a switch body or loop body.

Semantics

2 A **break** statement terminates execution of the smallest enclosing **switch** or iteration statement.

159) Following the **contin:** label is a null statement.

6.8.6.4 The `return` statement

Constraints

1 A **`return`** statement with an expression shall not appear in a function whose return type is **`void`**. A **`return`** statement without an expression shall only appear in a function whose return type is **`void`**.

Semantics

2 A **`return`** statement terminates execution of the current function and returns control to its caller. A function may have any number of **`return`** statements.

3 If a **`return`** statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.[160)]

4 EXAMPLE In:

```
struct s { double i; } f(void);
union {
      struct {
            int f1;
            struct s f2;
      } u1;
      struct {
            struct s f3;
            int f4;
      } u2;
} g;

struct s f(void)
{
      return g.u1.f2;
}

/* ... */
g.u2.f3 = f();
```

there is no undefined behavior, although there would be if the assignment were done directly (without using a function call to fetch the value).

160) The **`return`** statement is not an assignment. The overlap restriction of subclause 6.5.16.1 does not apply to the case of function return. The representation of floating-point values may have wider range or precision than implied by the type; a cast may be used to remove this extra range and precision.

6.9 External definitions

Syntax

1 *translation-unit:*
external-declaration
translation-unit external-declaration

external-declaration:
function-definition
declaration

Constraints

2 The storage-class specifiers **`auto`** and **`register`** shall not appear in the declaration specifiers in an external declaration.

3 There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. Moreover, if an identifier declared with internal linkage is used in an expression (other than as a part of the operand of a **`sizeof`** or **`_Alignof`** operator whose result is an integer constant), there shall be exactly one external definition for the identifier in the translation unit.

Semantics

4 As discussed in 5.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations. These are described as ‘‘external’’ because they appear outside any function (and hence have file scope). As discussed in 6.7, a declaration that also causes storage to be reserved for an object or a function named by the identifier is a definition.

5 An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **`sizeof`** or **`_Alignof`** operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.[161)]

161) Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.

6.9.1 Function definitions

Syntax

1 *function-definition:*
declaration-specifiers declarator declaration-list$_{opt}$ *compound-statement*

declaration-list:
declaration
declaration-list declaration

Constraints

2 The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.[162)]

3 The return type of a function shall be **`void`** or a complete object type other than array type.

4 The storage-class specifier, if any, in the declaration specifiers shall be either **`extern`** or **`static`**.

5 If the declarator includes a parameter type list, the declaration of each parameter shall include an identifier, except for the special case of a parameter list consisting of a single parameter of type **`void`**, in which case there shall not be an identifier. No declaration list shall follow.

6 If the declarator includes an identifier list, each declaration in the declaration list shall have at least one declarator, those declarators shall declare only identifiers from the identifier list, and every identifier in the identifier list shall be declared. An identifier declared as a typedef name shall not be redeclared as a parameter. The declarations in the declaration list shall contain no storage-class specifier other than **`register`** and no initializations.

---

162) The intent is that the type category in a function definition cannot be inherited from a typedef:

```
typedef int F(void);                 // type F is "function with no parameters
                                     //                  returning int"
F f, g;                              // f and g both have type compatible with F
F f { /* ... */ }                    // WRONG: syntax/constraint error
F g() { /* ... */ }                  // WRONG: declares that g returns a function
int f(void) { /* ... */ }            // RIGHT: f has type compatible with F
int g() { /* ... */ }                // RIGHT: g has type compatible with F
F *e(void) { /* ... */ }             // e returns a pointer to a function
F *((e))(void) { /* ... */ }         // same: parentheses irrelevant
int (*fp)(void);                     // fp points to a function that has type F
F *Fp;                               // Fp points to a function that has type F
```

Semantics

7 The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters. If the declarator includes a parameter type list, the list also specifies the types of all the parameters; such a declarator also serves as a function prototype for later calls to the same function in the same translation unit. If the declarator includes an identifier list,[163] the types of the parameters shall be declared in a following declaration list. In either case, the type of each parameter is adjusted as described in 6.7.6.3 for a parameter type list; the resulting type shall be a complete object type.

8 If a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation, the behavior is undefined.

9 Each parameter has automatic storage duration; its identifier is an lvalue.[164] The layout of the storage for parameters is unspecified.

10 On entry to the function, the size expressions of each variably modified parameter are evaluated and the value of each argument expression is converted to the type of the corresponding parameter as if by assignment. (Array expressions and function designators as arguments were converted to pointers before the call.)

11 After all parameters have been assigned, the compound statement that constitutes the body of the function definition is executed.

12 If the **}** that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined.

13 EXAMPLE 1 In the following:

```
extern int max(int a, int b)
{
      return a > b ? a : b;
}
```

`extern` is the storage-class specifier and **`int`** is the type specifier; **`max(int a, int b)`** is the function declarator; and

```
{ return a > b ? a : b; }
```

is the function body. The following similar definition uses the identifier-list form for the parameter declarations:

163) See ‘‘future language directions’’ (6.11.7).

164) A parameter identifier cannot be redeclared in the function body except in an enclosed block.

```
extern int max(a, b)
int a, b;
{
      return a > b ? a : b;
}
```

Here **int a, b;** is the declaration list for the parameters. The difference between these two definitions is that the first form acts as a prototype declaration that forces conversion of the arguments of subsequent calls to the function, whereas the second form does not.

14 EXAMPLE 2 To pass one function to another, one might say

```
      int f(void);
      /* ... */
      g(f);
```

Then the definition of **g** might read

```
void g(int (*funcp)(void))
{
      /* ... */
      (*funcp)(); /* or funcp(); ...  */
}
```

or, equivalently,

```
void g(int func(void))
{
      /* ... */
      func(); /* or (*func)(); ...  */
}
```

6.9.2 External object definitions

Semantics

1 If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition for the identifier.

2 A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier **static**, constitutes a *tentative definition*. If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to 0.

3 If the declaration of an identifier for an object is a tentative definition and has internal linkage, the declared type shall not be an incomplete type.

4 EXAMPLE 1

```
int i1 = 1;             // definition, external linkage
static int i2 = 2;      // definition, internal linkage
extern int i3 = 3;      // definition, external linkage
int i4;                 // tentative definition, external linkage
static int i5;          // tentative definition, internal linkage

int i1;                 // valid tentative definition, refers to previous
int i2;                 // 6.2.2 renders undefined, linkage disagreement
int i3;                 // valid tentative definition, refers to previous
int i4;                 // valid tentative definition, refers to previous
int i5;                 // 6.2.2 renders undefined, linkage disagreement

extern int i1;          // refers to previous, whose linkage is external
extern int i2;          // refers to previous, whose linkage is internal
extern int i3;          // refers to previous, whose linkage is external
extern int i4;          // refers to previous, whose linkage is external
extern int i5;          // refers to previous, whose linkage is internal
```

5 EXAMPLE 2 If at the end of the translation unit containing

```
int i[];
```

the array **i** still has incomplete type, the implicit initializer causes it to have one element, which is set to zero on program startup.

6.10 Preprocessing directives

Syntax

1 *preprocessing-file:*
  group$_{opt}$

group:
  group-part
  group group-part

group-part:
  if-section
  control-line
  text-line
  # *non-directive*

if-section:
  if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:
  # if *constant-expression new-line group*$_{opt}$
  # ifdef *identifier new-line group*$_{opt}$
  # ifndef *identifier new-line group*$_{opt}$

elif-groups:
  elif-group
  elif-groups elif-group

elif-group:
  # elif *constant-expression new-line group*$_{opt}$

else-group:
  # else *new-line group*$_{opt}$

endif-line:
  # endif *new-line*

control-line:
 # include *pp-tokens new-line*
 # define *identifier replacement-list new-line*
 # define *identifier lparen identifier-list$_{opt}$* **)**
replacement-list new-line
 # define *identifier lparen* **...)** *replacement-list new-line*
 # define *identifier lparen identifier-list* **, ...)**
replacement-list new-line
 # undef *identifier new-line*
 # line *pp-tokens new-line*
 # error *pp-tokens$_{opt}$ new-line*
 # pragma *pp-tokens$_{opt}$ new-line*
*new-line*

text-line:
pp-tokens$_{opt}$ new-line

non-directive:
pp-tokens new-line

lparen:
a **(** character not immediately preceded by white-space

replacement-list:
pp-tokens$_{opt}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

new-line:
the new-line character

Description

2 A *preprocessing directive* consists of a sequence of preprocessing tokens that satisfies the following constraints: The first token in the sequence is a **#** preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character. The last token in the sequence is the first new-line character that follows the first token in the sequence.[165] A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an

165) Thus, preprocessing directives are commonly called ''lines''. These ''lines'' have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the **#** character string literal creation operator in 6.10.3.2, for example).

invocation of a function-like macro.

3 A text line shall not begin with a **#** preprocessing token. A non-directive shall not begin with any of the directive names appearing in the syntax.

4 When in a group that is skipped (6.10.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

Constraints

5 The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing **#** preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

Semantics

6 The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

7 The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

8 EXAMPLE In:

```
#define EMPTY
EMPTY # include <file.h>
```

the sequence of preprocessing tokens on the second line is *not* a preprocessing directive, because it does not begin with a **#** at the start of translation phase 4, even though it will do so after the macro **EMPTY** has been replaced.

6.10.1 Conditional inclusion

Constraints

1 The expression that controls conditional inclusion shall be an integer constant expression except that: identifiers (including those lexically identical to keywords) are interpreted as described below;[166)] and it may contain unary operator expressions of the form

defined *identifier*

or

defined (*identifier* **)**

which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is

166) Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, etc.

predefined or if it has been the subject of a **#define** preprocessing directive without an intervening **#undef** directive with the same subject identifier), 0 if it is not.

2 Each preprocessing token that remains (in the list of preprocessing tokens that will become the controlling expression) after all macro replacements have occurred shall be in the lexical form of a 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 (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>**.[167)] 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.[168)] Also, whether a single-character character constant may have a negative value is implementation-defined.

---

167) 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.

168) 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)
```

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. 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.[169)]

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

6.10.2 Source file inclusion

Constraints

1 A **`#include`** directive shall identify a header or source file that can be processed by the implementation.

Semantics

2 A preprocessing directive of the form

> **`# include`** **<**_h-char-sequence_**>** *new-line*

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the **`<`** and **`>`** delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

3 A preprocessing directive of the form

> **`# include`** **"**_q-char-sequence_**"** *new-line*

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the **`"`** delimiters. The named source file is searched

---

169) As indicated by the syntax, a preprocessing token shall not follow a **`#else`** or **`#endif`** directive before the terminating new-line character. However, comments may appear anywhere in a source file, including within a preprocessing directive.

for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

`# include` **`<`***h-char-sequence***`>`** *new-line*

with the identical contained sequence (including **`>`** characters, if any) from the original directive.

4 A preprocessing directive of the form

`# include` *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after **`include`** in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements shall match one of the two previous forms.[170] The method by which a sequence of preprocessing tokens between a **`<`** and a **`>`** preprocessing token pair or a pair of **`"`** characters is combined into a single header name preprocessing token is implementation-defined.

5 The implementation shall provide unique mappings for sequences consisting of one or more nondigits or digits (6.4.2.1) followed by a period (**`.`**) and a single nondigit. The first character shall not be a digit. The implementation may ignore distinctions of alphabetical case and restrict the mapping to eight significant characters before the period.

6 A **`#include`** preprocessing directive may appear in a source file that has been read because of a **`#include`** directive in another file, up to an implementation-defined nesting limit (see 5.2.4.1).

7 EXAMPLE 1 The most common uses of **`#include`** preprocessing directives are as in the following:

```
#include <stdio.h>
#include "myprog.h"
```

170) Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 5.1.1.2); thus, an expansion that results in two string literals is an invalid directive.

8 EXAMPLE 2 This illustrates macro-replaced **#include** directives:

```
#if VERSION == 1
      #define INCFILE  "vers1.h"
#elif VERSION == 2
      #define INCFILE  "vers2.h"   // and so on
#else
      #define INCFILE  "versN.h"
#endif
#include INCFILE
```

Forward references: macro replacement (6.10.3).

6.10.3 Macro replacement

Constraints

1 Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

2 An identifier currently defined as an object-like macro shall not be redefined by another **#define** preprocessing directive unless the second definition is an object-like macro definition and the two replacement lists are identical. Likewise, an identifier currently defined as a function-like macro shall not be redefined by another **#define** preprocessing directive unless the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical.

3 There shall be white-space between the identifier and the replacement list in the definition of an object-like macro.

4 If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be more arguments in the invocation than there are parameters in the macro definition (excluding the **...**). There shall exist a **)** preprocessing token that terminates the invocation.

5 The identifier **__VA_ARGS__** shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters.

6 A parameter identifier in a function-like macro shall be uniquely declared within its scope.

Semantics

7 The identifier immediately following the **define** is called the *macro name*. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list

for either form of macro.

8 If a **#** preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

9 A preprocessing directive of the form

> **# define** *identifier replacement-list new-line*

defines an *object-like macro* that causes each subsequent instance of the macro name[171] to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. The replacement list is then rescanned for more macro names as specified below.

10 A preprocessing directive of the form

> **# define** *identifier lparen identifier-list*$_{opt}$ **)** *replacement-list new-line*
> **# define** *identifier lparen* **...)** *replacement-list new-line*
> **# define** *identifier lparen identifier-list* **, ...)** *replacement-list new-line*

defines a *function-like macro* with parameters, whose use is similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the **#define** preprocessing directive. Each subsequent instance of the function-like macro name followed by a **(** as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching **)** preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

11 The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives,[172] the behavior is undefined.

12 If there is a **...** in the identifier-list in the macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item:

---

171) Since, by macro-replacement time, all character constants and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 5.1.1.2, translation phases), they are never scanned for macro names or parameters.

172) Despite the name, a non-directive is a preprocessing directive.

the *variable arguments*. The number of arguments so combined is such that, following merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the `...`).

6.10.3.1 Argument substitution

1 After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a `#` or `##` preprocessing token or followed by a `##` preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the preprocessing file; no other preprocessing tokens are available.

2 An identifier `__VA_ARGS__` that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it.

6.10.3.2 The # operator

Constraints

1 Each `#` preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

Semantics

2 If, in the replacement list, a parameter is immediately preceded by a `#` preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token composing the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants: a `\` character is inserted before each `"` and `\` character of a character constant or string literal (including the delimiting `"` characters), except that it is implementation-defined whether a `\` character is inserted before the `\` character beginning a universal character name. If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty argument is `""`. The order of evaluation of `#` and `##` operators is unspecified.

6.10.3.3 The `##` operator

Constraints

1 A `##` preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

Semantics

2 If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a `##` preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a *placemarker* preprocessing token instead.[173)]

3 For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a `##` preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of `##` operators is unspecified.

4 EXAMPLE In the following fragment:

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)

char p[] = join(x, y); // equivalent to
                       // char p[] = "x ## y";
```

The expansion produces, at various stages:

```
join(x, y)

in_between(x hash_hash y)

in_between(x ## y)

mkstr(x ## y)

"x ## y"
```

In other words, expanding `hash_hash` produces a new token, consisting of two adjacent sharp signs, but this new token is not the `##` operator.

---

173) Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.

6.10.3.4 Rescanning and further replacement

1 After all parameters in the replacement list have been substituted and **#** and **##** processing has taken place, all placemarker preprocessing tokens are removed. The resulting preprocessing token sequence is then rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace.

2 If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

3 The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 6.10.9 below.

4 EXAMPLE There are cases where it is not clear whether a replacement is nested or not. For example, given the following macro definitions:

```
#define f(a) a*g
#define g(a) f(a)
```

the invocation

```
f(2)(9)
```

may expand to either

```
2*f(9)
```

or

```
2*9*g
```

Strictly conforming programs are not permitted to depend on such unspecified behavior.

6.10.3.5 Scope of macro definitions

1 A macro definition lasts (independent of block structure) until a corresponding **#undef** directive is encountered or (if none is encountered) until the end of the preprocessing translation unit. Macro definitions have no significance after translation phase 4.

2 A preprocessing directive of the form

 # undef *identifier new-line*

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

3 EXAMPLE 1 The simplest use of this facility is to define a ''manifest constant'', as in

```
#define TABSIZE 100
```

```
int table[TABSIZE];
```

4 EXAMPLE 2 The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.

5 EXAMPLE 3 To illustrate the rules for redefinition and reexamination, the sequence

```
#define x      3
#define f(a)   f(x * (a))
#undef  x
#define x      2
#define g      f
#define z      z[0]
#define h      g(~
#define m(a)   a(w)
#define w      0,1
#define t(a)   a
#define p()    int
#define q(x)   x
#define r(x,y) x ## y
#define str(x) # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
      (f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
char c[2][6] = { str(hello), str() };
```

results in

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1);
int i[] = { 1, 23, 4, 5,  };
char c[2][6] = { "hello", "" };
```

6 EXAMPLE 4 To illustrate the rules for creating character string literals and concatenating tokens, the sequence

```
#define str(s)      # s
#define xstr(s)     str(s)
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s", \
                        x ## s, x ## t)
#define INCFILE(n)  vers ## n
#define glue(a, b)  a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW     "hello"
#define LOW         LOW ", world"
```

```
debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4') // this goes away
      == 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in

```
printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
fputs(
  "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n",
  s);
#include "vers2.h"      (after macro replacement, before file access)
"hello";
"hello" ", world"
```

or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);
fputs(
  "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n",
  s);
#include "vers2.h"      (after macro replacement, before file access)
"hello";
"hello, world"
```

Space around the `#` and `##` tokens in the macro definition is optional.

7 EXAMPLE 5 To illustrate the rules for placemarker preprocessing tokens, the sequence

```
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,),
            t(10,,), t(,11,), t(,,12), t(,,) };
```

results in

```
int j[] = { 123, 45, 67, 89,
            10, 11, 12,  };
```

8 EXAMPLE 6 To demonstrate the redefinition rules, the following sequence is valid.

```
#define OBJ_LIKE      (1-1)
#define OBJ_LIKE      /* white space */ (1-1) /* other */
#define FUNC_LIKE(a)   ( a )
#define FUNC_LIKE( a )( /* note the white space */ \
                        a /* other stuff on this line
                           */ )
```

But the following redefinitions are invalid:

```
#define OBJ_LIKE    (0)     // different token sequence
#define OBJ_LIKE    (1 - 1) // different white space
#define FUNC_LIKE(b) ( a )  // different parameter usage
#define FUNC_LIKE(b) ( b )  // different parameter spelling
```

9 EXAMPLE 7 Finally, to show the variable argument list macro facilities:

```
#define debug(...)       fprintf(stderr, __VA_ARGS__)
#define showlist(...)    puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
            printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

results in

```
fprintf(stderr,  "Flag" );
fprintf(stderr,  "X = %d\n", x );
puts( "The first, second, and third items." );
((x>y)?puts("x>y"):
            printf("x is %d but y is %d", x, y));
```

6.10.4 Line control

Constraints

1 The string literal of a **`#line`** directive, if present, shall be a character string literal.

Semantics

2 The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (5.1.1.2) while processing the source file to the current token.

3 A preprocessing directive of the form

`# line` *digit-sequence new-line*

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). The digit sequence shall not specify zero, nor a number greater than 2147483647.

4 A preprocessing directive of the form

`# line` *digit-sequence* **"***s-char-sequence*$_{opt}$**"** *new-line*

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

5 A preprocessing directive of the form

`# line` *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after **`line`** on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). The directive resulting after all replacements shall match one of the two previous forms and is then processed as appropriate.

6.10.5 Error directive

Semantics

1 A preprocessing directive of the form

`# error` *pp-tokens*$_{opt}$ *new-line*

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens.

6.10.6 Pragma directive

Semantics

1 A preprocessing directive of the form

`# pragma` *pp-tokens*$_{opt}$ *new-line*

where the preprocessing token **`STDC`** does not immediately follow **`pragma`** in the directive (prior to any macro replacement)[174] causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any such **`pragma`** that is not recognized by the implementation is ignored.

2 If the preprocessing token **`STDC`** does immediately follow **`pragma`** in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms[175] whose meanings are described elsewhere:

`#pragma STDC FP_CONTRACT` *on-off-switch*
`#pragma STDC FENV_ACCESS` *on-off-switch*
`#pragma STDC CX_LIMITED_RANGE` *on-off-switch*

on-off-switch: one of
`ON   OFF   DEFAULT`

Forward references: the **`FP_CONTRACT`** pragma (7.12.2), the **`FENV_ACCESS`** pragma (7.6.1), the **`CX_LIMITED_RANGE`** pragma (7.3.4).

---

174) An implementation is not required to perform macro replacement in pragmas, but it is permitted except for in standard pragmas (where **`STDC`** immediately follows **`pragma`**). If the result of macro replacement in a non-standard pragma has the same form as a standard pragma, the behavior is still implementation-defined; an implementation is permitted to behave as if it were the standard pragma, but is not required to.

175) See ‘‘future language directions’’ (6.11.8).

6.10.7 Null directive

Semantics

1 A preprocessing directive of the form

*new-line*

has no effect.

6.10.8 Predefined macro names

1 The values of the predefined macros listed in the following subclauses[176)] (except for **__FILE__** and **__LINE__**) remain constant throughout the translation unit.

2 None of these macro names, nor the identifier **defined**, shall be the subject of a **#define** or a **#undef** preprocessing directive. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

3 The implementation shall not predefine the macro **__cplusplus**, nor shall it define it in any standard header.

Forward references: standard headers (7.1.2).

6.10.8.1 Mandatory macros

1 The following macro names shall be defined by the implementation:

__DATE__ The date of translation of the preprocessing translation unit: a character string literal of the form **"Mmm dd yyyy"**, where the names of the months are the same as those generated by the **asctime** function, and the first character of **dd** is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

__FILE__ The presumed name of the current source file (a character string literal).[177)]

__LINE__ The presumed line number (within the current source file) of the current source line (an integer constant).[177)]

__STDC__ The integer constant **1**, intended to indicate a conforming implementation.

__STDC_HOSTED__ The integer constant **1** if the implementation is a hosted implementation or the integer constant **0** if it is not.

176) See ‘‘future language directions’’ (6.11.9).

177) The presumed source file name and line number can be changed by the **#line** directive.

`__STDC_VERSION__` The integer constant **`201`***ymm***`L`**.[178)]

`__TIME__` The time of translation of the preprocessing translation unit: a character string literal of the form **`"hh:mm:ss"`** as in the time generated by the **`asctime`** function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

Forward references: the **`asctime`** function (7.27.3.1).

6.10.8.2 Environment macros

1 The following macro names are conditionally defined by the implementation:

`__STDC_ISO_10646__` An integer constant of the form **`yyyymmL`** (for example, **`199712L`**). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type **`wchar_t`**, has the same value as the short identifier of that character. The *Unicode required set* consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

`__STDC_MB_MIGHT_NEQ_WC__` The integer constant **`1`**, intended to indicate that, in the encoding for **`wchar_t`**, a member of the basic character set need not have a code value equal to its value when used as the lone character in an integer character constant.

`__STDC_UTF_16__` The integer constant **`1`**, intended to indicate that values of type **`char16_t`** are UTF−16 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

`__STDC_UTF_32__` The integer constant **`1`**, intended to indicate that values of type **`char32_t`** are UTF−32 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

Forward references: common definitions (7.19), unicode utilities (7.28).

178) This macro was not specified in ISO/IEC 9899:1990 and was specified as **`199409L`** in ISO/IEC 9899/AMD1:1995 and as **`199901L`** in ISO/IEC 9899:1999. The intention is that this will remain an integer constant of type **`long int`** that is increased with each revision of this International Standard.

6.10.8.3 Conditional feature macros

1 The following macro names are conditionally defined by the implementation:

`__STDC_ANALYZABLE__` The integer constant **`1`**, intended to indicate conformance to the specifications in annex L (Analyzability).

`__STDC_IEC_559__` The integer constant **`1`**, intended to indicate conformance to the specifications in annex F (IEC 60559 floating-point arithmetic).

`__STDC_IEC_559_COMPLEX__` The integer constant **`1`**, intended to indicate adherence to the specifications in annex G (IEC 60559 compatible complex arithmetic).

`__STDC_LIB_EXT1__` The integer constant **`201`***ymm***`L`**, intended to indicate support for the extensions defined in annex K (Bounds-checking interfaces).[179)]

`__STDC_NO_ATOMICS__` The integer constant **`1`**, intended to indicate that the implementation does not support atomic types (including the **`_Atomic`** type qualifier) and the **`<stdatomic.h>`** header.

`__STDC_NO_COMPLEX__` The integer constant **`1`**, intended to indicate that the implementation does not support complex types or the **`<complex.h>`** header.

`__STDC_NO_THREADS__` The integer constant **`1`**, intended to indicate that the implementation does not support the **`<threads.h>`** header.

`__STDC_NO_VLA__` The integer constant **`1`**, intended to indicate that the implementation does not support variable length arrays or variably modified types.

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

179) The intention is that this will remain an integer constant of type **`long int`** that is increased with each revision of this International Standard.

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 )
```

6.11 Future language directions

6.11.1 Floating types

1 Future standardization may include additional floating-point types, including those with greater range, precision, or both than `long double`.

6.11.2 Linkages of identifiers

1 Declaring an identifier with internal linkage at file scope without the `static` storage-class specifier is an obsolescent feature.

6.11.3 External names

1 Restriction of the significance of an external name to fewer than 255 characters (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations.

6.11.4 Character escape sequences

1 Lowercase letters as escape sequences are reserved for future standardization. Other characters may be used in extensions.

6.11.5 Storage-class specifiers

1 The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature.

6.11.6 Function declarators

1 The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.

6.11.7 Function definitions

1 The use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature.

6.11.8 Pragma directives

1 Pragmas whose first preprocessing token is `STDC` are reserved for future standardization.

6.11.9 Predefined macro names

1 Macro names beginning with `__STDC_` are reserved for future standardization.

7. Library

7.1 Introduction

7.1.1 Definitions of terms

1 A *string* is a contiguous sequence of characters terminated by and including the first null character. The term *multibyte string* is sometimes used instead to emphasize special processing given to multibyte characters contained in the string or to avoid confusion with a wide string. A *pointer to a string* is a pointer to its initial (lowest addressed) character. The *length of a string* is the number of bytes preceding the null character and the *value of a string* is the sequence of the values of the contained characters, in order.

2 The *decimal-point character* is the character used by functions that convert floating-point numbers to or from character sequences to denote the beginning of the fractional part of such character sequences.[180)] It is represented in the text and examples by a period, but may be changed by the **`setlocale`** function.

3 A *null wide character* is a wide character with code value zero.

4 A *wide string* is a contiguous sequence of wide characters terminated by and including the first null wide character. A *pointer to a wide string* is a pointer to its initial (lowest addressed) wide character. The *length of a wide string* is the number of wide characters preceding the null wide character and the *value of a wide string* is the sequence of code values of the contained wide characters, in order.

5 A *shift sequence* is a contiguous sequence of bytes within a multibyte string that (potentially) causes a change in shift state (see 5.2.1.2). A shift sequence shall not have a corresponding wide character; it is instead taken to be an adjunct to an adjacent multibyte character.[181)]

Forward references: character handling (7.4), the **`setlocale`** function (7.11.1.1).

180) The functions that make use of the decimal-point character are the numeric conversion functions (7.22.1, 7.29.4.1) and the formatted input/output functions (7.21.6, 7.29.2).

181) For state-dependent encodings, the values for **`MB_CUR_MAX`** and **`MB_LEN_MAX`** shall thus be large enough to count all the bytes in any complete multibyte character plus at least one adjacent shift sequence of maximum length. Whether these counts provide for more than one shift sequence is the implementation's choice.

7.1.2 Standard headers

1 Each library function is declared, with a type that includes a prototype, in a *header*,[182] whose contents are made available by the **#include** preprocessing directive. The header declares a set of related functions, plus any necessary types and additional macros needed to facilitate their use. Declarations of types described in this clause shall not include type qualifiers, unless explicitly stated otherwise.

2 The standard headers are[183]

<assert.h>	**<math.h>**	**<stdlib.h>**
<complex.h>	**<setjmp.h>**	**<stdnoreturn.h>**
<ctype.h>	**<signal.h>**	**<string.h>**
<errno.h>	**<stdalign.h>**	**<tgmath.h>**
<fenv.h>	**<stdarg.h>**	**<threads.h>**
<float.h>	**<stdatomic.h>**	**<time.h>**
<inttypes.h>	**<stdbool.h>**	**<uchar.h>**
<iso646.h>	**<stddef.h>**	**<wchar.h>**
<limits.h>	**<stdint.h>**	**<wctype.h>**
<locale.h>	**<stdio.h>**	

3 If a file with the same name as one of the above **<** and **>** delimited sequences, not provided as part of the implementation, is placed in any of the standard places that are searched for included source files, the behavior is undefined.

4 Standard headers may be included in any order; each may be included more than once in a given scope, with no effect different from being included only once, except that the effect of including **<assert.h>** depends on the definition of **NDEBUG** (see 7.2). If used, a header shall be included outside of any external declaration or definition, and it shall first be included before the first reference to any of the functions or objects it declares, or to any of the types or macros it defines. However, if an identifier is declared or defined in more than one header, the second and subsequent associated headers may be included after the initial reference to the identifier. The program shall not have any macros with names lexically identical to keywords currently defined prior to the inclusion of the header or when any macro defined in the header is expanded.

5 Any definition of an object-like macro described in this clause shall expand to code that is fully protected by parentheses where necessary, so that it groups in an arbitrary expression as if it were a single identifier.

182) A header is not necessarily a source file, nor are the **<** and **>** delimited sequences in header names necessarily valid source file names.

183) The headers **<complex.h>**, **<stdatomic.h>**, and **<threads.h>** are conditional features that implementations need not support; see 6.10.8.3.

6 Any declaration of a library function shall have external linkage.

7 A summary of the contents of the standard headers is given in annex B.

Forward references: diagnostics (7.2).

7.1.3 Reserved identifiers

1 Each header declares or defines all identifiers listed in its associated subclause, and optionally declares or defines identifiers listed in its associated future library directions subclause and identifiers which are always reserved either for any use or for use as file scope identifiers.

— All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use.

— All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary and tag name spaces.

— Each macro name in any of the following subclauses (including the future library directions) is reserved for use as specified if any of its associated headers is included; unless explicitly stated otherwise (see 7.1.4).

— All identifiers with external linkage in any of the following subclauses (including the future library directions) and **`errno`** are always reserved for use as identifiers with external linkage.[184)]

— Each identifier with file scope listed in any of the following subclauses (including the future library directions) is reserved for use as a macro name and as an identifier with file scope in the same name space if any of its associated headers is included.

2 No other identifiers are reserved. If the program declares or defines an identifier in a context in which it is reserved (other than as allowed by 7.1.4), or defines a reserved identifier as a macro name, the behavior is undefined.

3 If the program removes (with **`#undef`**) any macro definition of an identifier in the first group listed above, the behavior is undefined.

184) The list of reserved identifiers with external linkage includes **`math_errhandling`**, **`setjmp`**, **`va_copy`**, and **`va_end`**.

7.1.4 Use of library functions

1 Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow: If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer, or a pointer to non-modifiable storage when the corresponding parameter is not const-qualified) or a type (after promotion) not expected by a function with variable number of arguments, the behavior is undefined. If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid. Any function declared in a header may be additionally implemented as a function-like macro defined in the header, so if a library function is declared explicitly when its header is included, one of the techniques shown below can be used to ensure the declaration is not affected by such a macro. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro.[185)] The use of **#undef** to remove any macro definition will also ensure that an actual function is referred to. Any invocation of a library function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.[186)] Likewise, those function-like macros described in the following subclauses may be invoked in an expression anywhere a function with a compatible return type could be called.[187)] All object-like macros listed as expanding to

---

185) This means that an implementation shall provide an actual function for each library function, even if it also provides a macro for that function.

186) Such macros might not contain the sequence points that the corresponding function calls do.

187) Because external identifiers and some macro names beginning with an underscore are reserved, implementations may provide special semantics for such names. For example, the identifier **_BUILTIN_abs** could be used to indicate generation of in-line code for the **abs** function. Thus, the appropriate header could specify

```
#define abs(x) _BUILTIN_abs(x)
```

for a compiler whose code generator will accept it.

In this manner, a user desiring to guarantee that a given library function such as **abs** will be a genuine function may write

```
#undef abs
```

whether the implementation's header provides a macro implementation of **abs** or a built-in implementation. The prototype for the function, which precedes and is hidden by any macro definition, is thereby revealed also.

integer constant expressions shall additionally be suitable for use in **#if** preprocessing directives.

2 Provided that a library function can be declared without reference to any type defined in a header, it is also permissible to declare the function and use it without including its associated header.

3 There is a sequence point immediately before a library function returns.

4 The functions in the standard library are not guaranteed to be reentrant and may modify objects with static or thread storage duration.[188)]

5 Unless explicitly stated otherwise in the detailed descriptions that follow, library functions shall prevent data races as follows: A library function shall not directly or indirectly access objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's arguments. A library function shall not directly or indirectly modify objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's non-const arguments.[189)] Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

6 Unless otherwise specified, library functions shall perform all operations solely within the current thread if those operations have effects that are visible to users.[190)]

7 EXAMPLE The function **atoi** may be used in any of several ways:

— by use of its associated header (possibly generating a macro expansion)

```
#include <stdlib.h>
const char *str;
/* ... */
i = atoi(str);
```

— by use of its associated header (assuredly generating a true function reference)

188) Thus, a signal handler cannot, in general, call standard library functions.

189) This means, for example, that an implementation is not permitted to use a **static** object for internal purposes without synchronization because it could cause a data race even in programs that do not explicitly share objects between threads. Similarly, an implementation of **memcpy** is not permitted to copy bytes beyond the specified length of the destination object and then restore the original values because it could cause a data race if the program shared those bytes between threads.

190) This allows implementations to parallelize operations if there are no visible side effects.

```
#include <stdlib.h>
#undef atoi
const char *str;
/* ... */
i = atoi(str);
```

or

```
#include <stdlib.h>
const char *str;
/* ... */
i = (atoi)(str);
```

— by explicit declaration

```
extern int atoi(const char *);
const char *str;
/* ... */
i = atoi(str);
```

7.2 Diagnostics <assert.h>

1 The header **<assert.h>** defines the **assert** and **static_assert** macros and refers to another macro,

```
NDEBUG
```

which is *not* defined by **<assert.h>**. If **NDEBUG** is defined as a macro name at the point in the source file where **<assert.h>** is included, the **assert** macro is defined simply as

```
#define assert(ignore) ((void)0)
```

The **assert** macro is redefined according to the current state of **NDEBUG** each time that **<assert.h>** is included.

2 The **assert** macro shall be implemented as a macro, not as an actual function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

3 The macro

```
static_assert
```

expands to **_Static_assert**.

7.2.1 Program diagnostics

7.2.1.1 The assert macro

Synopsis

1
```
#include <assert.h>
void assert(scalar expression);
```

Description

2 The **assert** macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if **expression** (which shall have a scalar type) is false (that is, compares equal to 0), the **assert** macro writes information about the particular call that failed (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function — the latter are respectively the values of the preprocessing macros **__FILE__** and **__LINE__** and of the identifier **__func__**) on the standard error stream in an implementation-defined format.[191] It then calls the **abort** function.

---

191) The message written might be of the form:

```
Assertion failed: expression, function abc, file xyz, line nnn.
```

Returns

3 The `assert` macro returns no value.

Forward references: the `abort` function (7.22.4.1).

7.3 Complex arithmetic **<complex.h>**

7.3.1 Introduction

1 The header **<complex.h>** defines macros and declares functions that support complex arithmetic.[192)]

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

3 Each synopsis specifies a family of functions consisting of a principal function with one or more **double complex** parameters and a **double complex** or **double** return value; and other functions with the same name but with **f** and **l** suffixes which are corresponding functions with **float** and **long double** parameters and return values.

4 The macro

```
complex
```

expands to **_Complex**; the macro

```
_Complex_I
```

expands to a constant expression of type **const float _Complex**, with the value of the imaginary unit.[193)]

5 The macros

```
imaginary
```

and

```
_Imaginary_I
```

are defined if and only if the implementation supports imaginary types;[194)] if defined, they expand to **_Imaginary** and a constant expression of type **const float _Imaginary** with the value of the imaginary unit.

6 The macro

```
I
```

expands to either **_Imaginary_I** or **_Complex_I**. If **_Imaginary_I** is not defined, **I** shall expand to **_Complex_I**.

7 Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros **complex**, **imaginary**, and **I**.

---

192) See ‘‘future library directions’’ (7.31.1).

193) The imaginary unit is a number *i* such that $i^2 = -1$.

194) A specification for imaginary types is in informative annex G.

Forward references: IEC 60559-compatible complex arithmetic (annex G).

7.3.2 Conventions

1 Values are interpreted as radians, not degrees. An implementation may set **`errno`** but is not required to.

7.3.3 Branch cuts

1 Some of the functions below have branch cuts, across which the function is discontinuous. For implementations with a signed zero (including all IEC 60559 implementations) that follow the specifications of annex G, the sign of zero distinguishes one side of a cut from another so the function is continuous (except for format limitations) as the cut is approached from either side. For example, for the square root function, which has a branch cut along the negative real axis, the top of the cut, with imaginary part +0, maps to the positive imaginary axis, and the bottom of the cut, with imaginary part −0, maps to the negative imaginary axis.

2 Implementations that do not support a signed zero (see annex F) cannot distinguish the sides of branch cuts. These implementations shall map a cut so the function is continuous as the cut is approached coming around the finite endpoint of the cut in a counter clockwise direction. (Branch cuts for the functions specified here have just one finite endpoint.) For example, for the square root function, coming counter clockwise around the finite endpoint of the cut along the negative real axis approaches the cut from above, so the cut maps to the positive imaginary axis.

7.3.4 The `CX_LIMITED_RANGE` pragma

Synopsis

1
```
#include <complex.h>
#pragma STDC CX_LIMITED_RANGE on-off-switch
```

Description

2 The usual mathematical formulas for complex multiply, divide, and absolute value are problematic because of their treatment of infinities and because of undue overflow and underflow. The **`CX_LIMITED_RANGE`** pragma can be used to inform the implementation that (where the state is ‘‘on’’) the usual mathematical formulas are acceptable.[195)] The pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another **`CX_LIMITED_RANGE`** pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another **`CX_LIMITED_RANGE`** pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the

compound statement. If this pragma is used in any other context, the behavior is undefined. The default state for the pragma is ‘‘off’’.

7.3.5 Trigonometric functions

7.3.5.1 The `cacos` functions

Synopsis

1
```
#include <complex.h>
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

Description

2 The **`cacos`** functions compute the complex arc cosine of **`z`**, with branch cuts outside the interval $[-1, +1]$ along the real axis.

Returns

3 The **`cacos`** functions return the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[0, \pi]$ along the real axis.

7.3.5.2 The `casin` functions

Synopsis

1
```
#include <complex.h>
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

Description

2 The **`casin`** functions compute the complex arc sine of **`z`**, with branch cuts outside the interval $[-1, +1]$ along the real axis.

Returns

3 The **`casin`** functions return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$

---

195) The purpose of the pragma is to allow the implementation to use the formulas:

$$(x + iy) \times (u + iv) = (xu - yv) + i(yu + xv)$$

$$(x + iy) \ / \ (u + iv) = [(xu + yv) + i(yu - xv)]/(u^2 + v^2)$$

$$|\, x + iy \,| = \sqrt{x^2 + y^2}$$

where the programmer can determine they are safe.

along the real axis.

7.3.5.3 The `catan` functions

Synopsis

1
```
#include <complex.h>
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

Description

2 The **`catan`** functions compute the complex arc tangent of **`z`**, with branch cuts outside the interval $[-i, +i]$ along the imaginary axis.

Returns

3 The **`catan`** functions return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

7.3.5.4 The `ccos` functions

Synopsis

1
```
#include <complex.h>
double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
```

Description

2 The **`ccos`** functions compute the complex cosine of **`z`**.

Returns

3 The **`ccos`** functions return the complex cosine value.

7.3.5.5 The `csin` functions

Synopsis

1
```
#include <complex.h>
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

Description

2 The **`csin`** functions compute the complex sine of **`z`**.

Returns

3 The **`csin`** functions return the complex sine value.

7.3.5.6 The `ctan` functions

Synopsis

1
```
#include <complex.h>
double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

Description

2 The **`ctan`** functions compute the complex tangent of **`z`**.

Returns

3 The **`ctan`** functions return the complex tangent value.

7.3.6 Hyperbolic functions

7.3.6.1 The `cacosh` functions

Synopsis

1
```
#include <complex.h>
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

Description

2 The **`cacosh`** functions compute the complex arc hyperbolic cosine of **`z`**, with a branch cut at values less than 1 along the real axis.

Returns

3 The **`cacosh`** functions return the complex arc hyperbolic cosine value, in the range of a half-strip of nonnegative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

7.3.6.2 The `casinh` functions

Synopsis

1
```
#include <complex.h>
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

Description

2 The **casinh** functions compute the complex arc hyperbolic sine of **z**, with branch cuts outside the interval $[-i, +i]$ along the imaginary axis.

Returns

3 The **casinh** functions return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.

7.3.6.3 The catanh functions

Synopsis

1
```
#include <complex.h>
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

Description

2 The **catanh** functions compute the complex arc hyperbolic tangent of **z**, with branch cuts outside the interval $[-1, +1]$ along the real axis.

Returns

3 The **catanh** functions return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.

7.3.6.4 The ccosh functions

Synopsis

1
```
#include <complex.h>
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

Description

2 The **ccosh** functions compute the complex hyperbolic cosine of **z**.

Returns

3 The **ccosh** functions return the complex hyperbolic cosine value.

7.3.6.5 The `csinh` functions

Synopsis

1
```
#include <complex.h>
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

Description

2 The **`csinh`** functions compute the complex hyperbolic sine of **`z`**.

Returns

3 The **`csinh`** functions return the complex hyperbolic sine value.

7.3.6.6 The `ctanh` functions

Synopsis

1
```
#include <complex.h>
double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

Description

2 The **`ctanh`** functions compute the complex hyperbolic tangent of **`z`**.

Returns

3 The **`ctanh`** functions return the complex hyperbolic tangent value.

7.3.7 Exponential and logarithmic functions

7.3.7.1 The `cexp` functions

Synopsis

1
```
#include <complex.h>
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
```

Description

2 The **`cexp`** functions compute the complex base-*e* exponential of **`z`**.

Returns

3 The **`cexp`** functions return the complex base-*e* exponential value.

7.3.7.2 The `clog` functions

Synopsis

1
```
#include <complex.h>
double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

Description

2 The `clog` functions compute the complex natural (base-*e*) logarithm of `z`, with a branch cut along the negative real axis.

Returns

3 The `clog` functions return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

7.3.8 Power and absolute-value functions

7.3.8.1 The `cabs` functions

Synopsis

1
```
#include <complex.h>
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
```

Description

2 The `cabs` functions compute the complex absolute value (also called norm, modulus, or magnitude) of `z`.

Returns

3 The `cabs` functions return the complex absolute value.

7.3.8.2 The `cpow` functions

Synopsis

1
```
#include <complex.h>
double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x,
     long double complex y);
```

Description

2 The **`cpow`** functions compute the complex power function $\mathbf{x}^{\mathbf{y}}$, with a branch cut for the first parameter along the negative real axis.

Returns

3 The **`cpow`** functions return the complex power function value.

7.3.8.3 The `csqrt` functions

Synopsis

1
```
#include <complex.h>
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

Description

2 The **`csqrt`** functions compute the complex square root of **`z`**, with a branch cut along the negative real axis.

Returns

3 The **`csqrt`** functions return the complex square root value, in the range of the right half-plane (including the imaginary axis).

7.3.9 Manipulation functions

7.3.9.1 The `carg` functions

Synopsis

1
```
#include <complex.h>
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

Description

2 The **`carg`** functions compute the argument (also called phase angle) of **`z`**, with a branch cut along the negative real axis.

Returns

3 The **`carg`** functions return the value of the argument in the interval $[-\pi, +\pi]$.

7.3.9.2 The `cimag` functions

Synopsis

1
```
#include <complex.h>
double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
```

Description

2 The `cimag` functions compute the imaginary part of `z`.[196)]

Returns

3 The `cimag` functions return the imaginary part value (as a real).

7.3.9.3 The `CMPLX` macros

Synopsis

1
```
#include <complex.h>
double complex CMPLX(double x, double y);
float complex CMPLXF(float x, float y);
long double complex CMPLXL(long double x, long double y);
```

Description

2 The `CMPLX` macros expand to an expression of the specified complex type, with the real part having the (converted) value of `x` and the imaginary part having the (converted) value of `y`. The resulting expression shall be suitable for use as an initializer for an object with static or thread storage duration, provided both arguments are likewise suitable. |

Returns

3 The `CMPLX` macros return the complex value `x` + *i* `y`.

4 NOTE These macros act as if the implementation supported imaginary types and the definitions were:

```
#define CMPLX(x, y)  ((double complex)((double)(x) + \
                                _Imaginary_I * (double)(y)))
#define CMPLXF(x, y) ((float complex)((float)(x) + \
                                _Imaginary_I * (float)(y)))
#define CMPLXL(x, y) ((long double complex)((long double)(x) + \
                                _Imaginary_I * (long double)(y)))
```

196) For a variable `z` of complex type, `z == creal(z) + cimag(z)*I`.

7.3.9.4 The `conj` functions

Synopsis

1
```
#include <complex.h>
double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

Description

2 The **conj** functions compute the complex conjugate of **z**, by reversing the sign of its imaginary part.

Returns

3 The **conj** functions return the complex conjugate value.

7.3.9.5 The `cproj` functions

Synopsis

1
```
#include <complex.h>
double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

Description

2 The **cproj** functions compute a projection of **z** onto the Riemann sphere: **z** projects to **z** except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If **z** has an infinite part, then **cproj(z)** is equivalent to

```
INFINITY + I * copysign(0.0, cimag(z))
```

Returns

3 The **cproj** functions return the value of the projection onto the Riemann sphere.

7.3.9.6 The `creal` functions

Synopsis

1
```
#include <complex.h>
double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

Description

2 The **creal** functions compute the real part of **z**.[197)]

Returns

3 The **creal** functions return the real part value.

197) For a variable **z** of complex type, **z == creal(z) + cimag(z)*I**.

7.4 Character handling `<ctype.h>`

1 The header **`<ctype.h>`** declares several functions useful for classifying and mapping characters.[198] In all cases the argument is an **`int`**, the value of which shall be representable as an **`unsigned char`** or shall equal the value of the macro **`EOF`**. If the argument has any other value, the behavior is undefined.

2 The behavior of these functions is affected by the current locale. Those functions that have locale-specific aspects only when not in the **`"C"`** locale are noted below.

3 The term *printing character* refers to a member of a locale-specific set of characters, each of which occupies one printing position on a display device; the term *control character* refers to a member of a locale-specific set of characters that are not printing characters.[199] All letters and digits are printing characters.

Forward references: **`EOF`** (7.21.1), localization (7.11).

7.4.1 Character classification functions

1 The functions in this subclause return nonzero (true) if and only if the value of the argument **`c`** conforms to that in the description of the function.

7.4.1.1 The `isalnum` function

Synopsis

1
```
#include <ctype.h>
int isalnum(int c);
```

Description

2 The **`isalnum`** function tests for any character for which **`isalpha`** or **`isdigit`** is true.

7.4.1.2 The `isalpha` function

Synopsis

1
```
#include <ctype.h>
int isalpha(int c);
```

Description

2 The **`isalpha`** function tests for any character for which **`isupper`** or **`islower`** is true, or any character that is one of a locale-specific set of alphabetic characters for which

198) See ‘‘future library directions’’ (7.31.2).

199) In an implementation that uses the seven-bit US ASCII character set, the printing characters are those whose values lie from 0x20 (space) through 0x7E (tilde); the control characters are those whose values lie from 0 (NUL) through 0x1F (US), and the character 0x7F (DEL).

none of **iscntrl**, **isdigit**, **ispunct**, or **isspace** is true.[200] In the **"C"** locale, **isalpha** returns true only for the characters for which **isupper** or **islower** is true.

7.4.1.3 The isblank function

Synopsis

1
```
#include <ctype.h>
int isblank(int c);
```

Description

2 The **isblank** function tests for any character that is a standard blank character or is one of a locale-specific set of characters for which **isspace** is true and that is used to separate words within a line of text. The standard blank characters are the following: space (**' '**), and horizontal tab (**'\t'**). In the **"C"** locale, **isblank** returns true only for the standard blank characters.

7.4.1.4 The iscntrl function

Synopsis

1
```
#include <ctype.h>
int iscntrl(int c);
```

Description

2 The **iscntrl** function tests for any control character.

7.4.1.5 The isdigit function

Synopsis

1
```
#include <ctype.h>
int isdigit(int c);
```

Description

2 The **isdigit** function tests for any decimal-digit character (as defined in 5.2.1).

7.4.1.6 The isgraph function

Synopsis

1
```
#include <ctype.h>
int isgraph(int c);
```

200) The functions **islower** and **isupper** test true or false separately for each of these additional characters; all four combinations are possible.

Description

2 The **`isgraph`** function tests for any printing character except space (**`' '`**).

7.4.1.7 The `islower` function

Synopsis

1
```
#include <ctype.h>
int islower(int c);
```

Description

2 The **`islower`** function tests for any character that is a lowercase letter or is one of a locale-specific set of characters for which none of **`iscntrl`**, **`isdigit`**, **`ispunct`**, or **`isspace`** is true. In the **`"C"`** locale, **`islower`** returns true only for the lowercase letters (as defined in 5.2.1).

7.4.1.8 The `isprint` function

Synopsis

1
```
#include <ctype.h>
int isprint(int c);
```

Description

2 The **`isprint`** function tests for any printing character including space (**`' '`**).

7.4.1.9 The `ispunct` function

Synopsis

1
```
#include <ctype.h>
int ispunct(int c);
```

Description

2 The **`ispunct`** function tests for any printing character that is one of a locale-specific set of punctuation characters for which neither **`isspace`** nor **`isalnum`** is true. In the **`"C"`** locale, **`ispunct`** returns true for every printing character for which neither **`isspace`** nor **`isalnum`** is true.

7.4.1.10 The `isspace` function

Synopsis

1
```
#include <ctype.h>
int isspace(int c);
```

Description

2 The **`isspace`** function tests for any character that is a standard white-space character or is one of a locale-specific set of characters for which **`isalnum`** is false. The standard

white-space characters are the following: space (`' '`), form feed (`'\f'`), new-line (`'\n'`), carriage return (`'\r'`), horizontal tab (`'\t'`), and vertical tab (`'\v'`). In the `"C"` locale, `isspace` returns true only for the standard white-space characters.

7.4.1.11 The `isupper` function

Synopsis

1
```
#include <ctype.h>
int isupper(int c);
```

Description

2 The `isupper` function tests for any character that is an uppercase letter or is one of a locale-specific set of characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true. In the `"C"` locale, `isupper` returns true only for the uppercase letters (as defined in 5.2.1).

7.4.1.12 The `isxdigit` function

Synopsis

1
```
#include <ctype.h>
int isxdigit(int c);
```

Description

2 The `isxdigit` function tests for any hexadecimal-digit character (as defined in 6.4.4.1).

7.4.2 Character case mapping functions

7.4.2.1 The `tolower` function

Synopsis

1
```
#include <ctype.h>
int tolower(int c);
```

Description

2 The `tolower` function converts an uppercase letter to a corresponding lowercase letter.

Returns

3 If the argument is a character for which `isupper` is true and there are one or more corresponding characters, as specified by the current locale, for which `islower` is true, the `tolower` function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.4.2.2 The `toupper` function

Synopsis

1
```
#include <ctype.h>
int toupper(int c);
```

Description

2 The **`toupper`** function converts a lowercase letter to a corresponding uppercase letter.

Returns

3 If the argument is a character for which **`islower`** is true and there are one or more corresponding characters, as specified by the current locale, for which **`isupper`** is true, the **`toupper`** function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.5 Errors `<errno.h>`

1 The header **`<errno.h>`** defines several macros, all relating to the reporting of error conditions.

2 The macros are

```
EDOM
EILSEQ
ERANGE
```

which expand to integer constant expressions with type **`int`**, distinct positive values, and which are suitable for use in **`#if`** preprocessing directives; and

```
errno
```

which expands to a modifiable lvalue[201)] that has type **`int`** and thread local storage duration, the value of which is set to a positive error number by several library functions. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name **`errno`**, the behavior is undefined.

3 The value of **`errno`** in the initial thread is zero at program startup (the initial value of **`errno`** in other threads is an indeterminate value), but is never set to zero by any library function.[202)] The value of **`errno`** may be set to nonzero by a library function call whether or not there is an error, provided the use of **`errno`** is not documented in the description of the function in this International Standard.

4 Additional macro definitions, beginning with **`E`** and a digit or **`E`** and an uppercase letter,[203)] may also be specified by the implementation.

---

201) The macro **`errno`** need not be the identifier of an object. It might expand to a modifiable lvalue resulting from a function call (for example, **`*errno()`**).

202) Thus, a program that uses **`errno`** for error checking should set it to zero before a library function call, then inspect it before a subsequent library function call. Of course, a library function can save the value of **`errno`** on entry and then set it to zero, as long as the original value is restored if **`errno`**'s value is still zero just before the return.

203) See ‘‘future library directions’’ (7.31.3).

7.6 Floating-point environment `<fenv.h>`

1 The header **`<fenv.h>`** defines several macros, and declares types and functions that provide access to the floating-point environment. The *floating-point environment* refers collectively to any floating-point status flags and control modes supported by the implementation.[204] A *floating-point status flag* is a system variable whose value is set (but never cleared) when a *floating-point exception* is raised, which occurs as a side effect of exceptional floating-point arithmetic to provide auxiliary information.[205] A *floating-point control mode* is a system variable whose value may be set by the user to affect the subsequent behavior of floating-point arithmetic.

2 The floating-point environment has thread storage duration. The initial state for a thread's floating-point environment is the current state of the floating-point environment of the thread that creates it at the time of creation.

3 Certain programming conventions support the intended model of use for the floating-point environment:[206]

— a function call does not alter its caller's floating-point control modes, clear its caller's floating-point status flags, nor depend on the state of its caller's floating-point status flags unless the function is so documented;

— a function call is assumed to require default floating-point control modes, unless its documentation promises otherwise;

— a function call is assumed to have the potential for raising floating-point exceptions, unless its documentation promises otherwise.

4 The type

```
fenv_t
```

represents the entire floating-point environment.

5 The type

```
fexcept_t
```

represents the floating-point status flags collectively, including any status the implementation associates with the flags.

---

204) This header is designed to support the floating-point exception status flags and directed-rounding control modes required by IEC 60559, and other similar floating-point state information. It is also designed to facilitate code portability among all systems.

205) A floating-point status flag is not an object and can be set more than once within an expression.

206) With these conventions, a programmer can safely assume default floating-point control modes (or be unaware of them). The responsibilities associated with accessing the floating-point environment fall on the programmer or program that does so explicitly.

6 Each of the macros

```
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
```

is defined if and only if the implementation supports the floating-point exception by means of the functions in 7.6.2.[207] Additional implementation-defined floating-point exceptions, with macro definitions beginning with **FE_** and an uppercase letter,[208] may also be specified by the implementation. The defined macros expand to integer constant expressions with values such that bitwise ORs of all combinations of the macros result in distinct values, and furthermore, bitwise ANDs of all combinations of the macros result in zero.[209]

7 The macro

```
FE_ALL_EXCEPT
```

is simply the bitwise OR of all floating-point exception macros defined by the implementation. If no such macros are defined, **FE_ALL_EXCEPT** shall be defined as **0**.

8 Each of the macros

```
FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD
```

is defined if and only if the implementation supports getting and setting the represented rounding direction by means of the **fegetround** and **fesetround** functions. Additional implementation-defined rounding directions, with macro definitions beginning with **FE_** and an uppercase letter,[210] may also be specified by the implementation. The defined macros expand to integer constant expressions whose values are distinct nonnegative values.[211]

207) The implementation supports a floating-point exception if there are circumstances where a call to at least one of the functions in 7.6.2, using the macro as the appropriate argument, will succeed. It is not necessary for all the functions to succeed all the time.

208) See ‘‘future library directions’’ (7.31.4).

209) The macros should be distinct powers of two.

210) See ‘‘future library directions’’ (7.31.4).

211) Even though the rounding direction macros may expand to constants corresponding to the values of **FLT_ROUNDS**, they are not required to do so.

9 The macro

```
FE_DFL_ENV
```

represents the default floating-point environment — the one installed at program startup — and has type ‘‘pointer to const-qualified **fenv_t**’’. It can be used as an argument to **<fenv.h>** functions that manage the floating-point environment.

10 Additional implementation-defined environments, with macro definitions beginning with **FE_** and an uppercase letter,[212] and having type ‘‘pointer to const-qualified **fenv_t**’’, may also be specified by the implementation.

7.6.1 The FENV_ACCESS pragma

Synopsis

1
```
#include <fenv.h>
#pragma STDC FENV_ACCESS on-off-switch
```

Description

2 The **FENV_ACCESS** pragma provides a means to inform the implementation when a program might access the floating-point environment to test floating-point status flags or run under non-default floating-point control modes.[213] The pragma shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another **FENV_ACCESS** pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another **FENV_ACCESS** pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. If part of a program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the **FENV_ACCESS** pragma ‘‘off’’, the behavior is undefined. The default state (‘‘on’’ or ‘‘off’’) for the pragma is implementation-defined. (When execution passes from a part of the program translated with **FENV_ACCESS** ‘‘off’’ to a part translated with **FENV_ACCESS** ‘‘on’’, the state of the floating-point status flags is unspecified and the floating-point control modes have their default settings.)

212) See ‘‘future library directions’’ (7.31.4).

213) The purpose of the **FENV_ACCESS** pragma is to allow certain optimizations that could subvert flag tests and mode changes (e.g., global common subexpression elimination, code motion, and constant folding). In general, if the state of **FENV_ACCESS** is ‘‘off’’, the translator can assume that default modes are in effect and the flags are not tested.

3 EXAMPLE

```
#include <fenv.h>
void f(double x)
{
      #pragma STDC FENV_ACCESS ON
      void g(double);
      void h(double);
      /* ... */
      g(x + 1);
      h(x + 1);
      /* ... */
}
```

4 If the function **g** might depend on status flags set as a side effect of the first **x + 1**, or if the second **x + 1** might depend on control modes set as a side effect of the call to function **g**, then the program shall contain an appropriately placed invocation of **#pragma STDC FENV_ACCESS ON**.[214)]

7.6.2 Floating-point exceptions

1 The following functions provide access to the floating-point status flags.[215)] The **int** input argument for the functions represents a subset of floating-point exceptions, and can be zero or the bitwise OR of one or more floating-point exception macros, for example **FE_OVERFLOW | FE_INEXACT**. For other argument values the behavior of these functions is undefined.

7.6.2.1 The `feclearexcept` function

Synopsis

1
```
#include <fenv.h>
int feclearexcept(int excepts);
```

Description

2 The **feclearexcept** function attempts to clear the supported floating-point exceptions represented by its argument.

Returns

3 The **feclearexcept** function returns zero if the **excepts** argument is zero or if all the specified exceptions were successfully cleared. Otherwise, it returns a nonzero value.

---

214) The side effects impose a temporal ordering that requires two evaluations of **x + 1**. On the other hand, without the **#pragma STDC FENV_ACCESS ON** pragma, and assuming the default state is ‘‘off’’, just one evaluation of **x + 1** would suffice.

215) The functions **fetestexcept**, **feraiseexcept**, and **feclearexcept** support the basic abstraction of flags that are either set or clear. An implementation may endow floating-point status flags with more information — for example, the address of the code which first raised the floating-point exception; the functions **fegetexceptflag** and **fesetexceptflag** deal with the full content of flags.

7.6.2.2 The `fegetexceptflag` function

Synopsis

1
```
#include <fenv.h>
int fegetexceptflag(fexcept_t *flagp,
     int excepts);
```

Description

2 The **`fegetexceptflag`** function attempts to store an implementation-defined representation of the states of the floating-point status flags indicated by the argument **`excepts`** in the object pointed to by the argument **`flagp`**.

Returns

3 The **`fegetexceptflag`** function returns zero if the representation was successfully stored. Otherwise, it returns a nonzero value.

7.6.2.3 The `feraiseexcept` function

Synopsis

1
```
#include <fenv.h>
int feraiseexcept(int excepts);
```

Description

2 The **`feraiseexcept`** function attempts to raise the supported floating-point exceptions represented by its argument.[216)] The order in which these floating-point exceptions are raised is unspecified, except as stated in F.8.6. Whether the **`feraiseexcept`** function additionally raises the ‘‘inexact’’ floating-point exception whenever it raises the ‘‘overflow’’ or ‘‘underflow’’ floating-point exception is implementation-defined.

Returns

3 The **`feraiseexcept`** function returns zero if the **`excepts`** argument is zero or if all the specified exceptions were successfully raised. Otherwise, it returns a nonzero value.

216) The effect is intended to be similar to that of floating-point exceptions raised by arithmetic operations. Hence, enabled traps for floating-point exceptions raised by this function are taken. The specification in F.8.6 is in the same spirit.

7.6.2.4 The `fesetexceptflag` function

Synopsis

1
```
#include <fenv.h>
int fesetexceptflag(const fexcept_t *flagp,
     int excepts);
```

Description

2 The **fesetexceptflag** function attempts to set the floating-point status flags indicated by the argument **excepts** to the states stored in the object pointed to by **flagp**. The value of ***flagp** shall have been set by a previous call to **fegetexceptflag** whose second argument represented at least those floating-point exceptions represented by the argument **excepts**. This function does not raise floating-point exceptions, but only sets the state of the flags.

Returns

3 The **fesetexceptflag** function returns zero if the **excepts** argument is zero or if all the specified flags were successfully set to the appropriate state. Otherwise, it returns a nonzero value.

7.6.2.5 The `fetestexcept` function

Synopsis

1
```
#include <fenv.h>
int fetestexcept(int excepts);
```

Description

2 The **fetestexcept** function determines which of a specified subset of the floating-point exception flags are currently set. The **excepts** argument specifies the floating-point status flags to be queried.[217)]

Returns

3 The **fetestexcept** function returns the value of the bitwise OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in **excepts**.

4 EXAMPLE Call **f** if ''invalid'' is set, then **g** if ''overflow'' is set:

217) This mechanism allows testing several floating-point exceptions with just one function call.

```
#include <fenv.h>
/* ... */
{
      #pragma STDC FENV_ACCESS ON
      int set_excepts;
      feclearexcept(FE_INVALID | FE_OVERFLOW);
      // maybe raise exceptions
      set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
      if (set_excepts & FE_INVALID) f();
      if (set_excepts & FE_OVERFLOW) g();
      /* ... */
}
```

7.6.3 Rounding

1 The **`fegetround`** and **`fesetround`** functions provide control of rounding direction modes.

7.6.3.1 The `fegetround` function

Synopsis

1
```
#include <fenv.h>
int fegetround(void);
```

Description

2 The **`fegetround`** function gets the current rounding direction.

Returns

3 The **`fegetround`** function returns the value of the rounding direction macro representing the current rounding direction or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.

7.6.3.2 The `fesetround` function

Synopsis

1
```
#include <fenv.h>
int fesetround(int round);
```

Description

2 The **`fesetround`** function establishes the rounding direction represented by its argument **`round`**. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

Returns

3 The **`fesetround`** function returns zero if and only if the requested rounding direction was established.

4 EXAMPLE Save, set, and restore the rounding direction. Report an error and abort if setting the rounding direction fails.

```
#include <fenv.h>
#include <assert.h>

void f(int round_dir)
{
      #pragma STDC FENV_ACCESS ON
      int save_round;
      int setround_ok;
      save_round = fegetround();
      setround_ok = fesetround(round_dir);
      assert(setround_ok == 0);
      /* ... */
      fesetround(save_round);
      /* ... */
}
```

7.6.4 Environment

1 The functions in this section manage the floating-point environment — status flags and control modes — as one entity.

7.6.4.1 The `fegetenv` function

Synopsis

1
```
#include <fenv.h>
int fegetenv(fenv_t *envp);
```

Description

2 The **`fegetenv`** function attempts to store the current floating-point environment in the object pointed to by **`envp`**.

Returns

3 The **`fegetenv`** function returns zero if the environment was successfully stored. Otherwise, it returns a nonzero value.

7.6.4.2 The `feholdexcept` function

Synopsis

1
```
#include <fenv.h>
int feholdexcept(fenv_t *envp);
```

Description

2 The **`feholdexcept`** function saves the current floating-point environment in the object pointed to by **`envp`**, clears the floating-point status flags, and then installs a *non-stop* (continue on floating-point exceptions) mode, if available, for all floating-point exceptions.[218)]

Returns

3 The **feholdexcept** function returns zero if and only if non-stop floating-point exception handling was successfully installed.

7.6.4.3 The fesetenv function

Synopsis

1
```
#include <fenv.h>
int fesetenv(const fenv_t *envp);
```

Description

2 The **fesetenv** function attempts to establish the floating-point environment represented by the object pointed to by **envp**. The argument **envp** shall point to an object set by a call to **fegetenv** or **feholdexcept**, or equal a floating-point environment macro. Note that **fesetenv** merely installs the state of the floating-point status flags represented through its argument, and does not raise these floating-point exceptions.

Returns

3 The **fesetenv** function returns zero if the environment was successfully established. Otherwise, it returns a nonzero value.

7.6.4.4 The feupdateenv function

Synopsis

1
```
#include <fenv.h>
int feupdateenv(const fenv_t *envp);
```

Description

2 The **feupdateenv** function attempts to save the currently raised floating-point exceptions in its automatic storage, install the floating-point environment represented by the object pointed to by **envp**, and then raise the saved floating-point exceptions. The argument **envp** shall point to an object set by a call to **feholdexcept** or **fegetenv**, or equal a floating-point environment macro.

Returns

3 The **feupdateenv** function returns zero if all the actions were successfully carried out. Otherwise, it returns a nonzero value.

---

218) IEC 60559 systems have a default non-stop mode, and typically at least one other mode for trap handling or aborting; if the system provides only the non-stop mode then installing it is trivial. For such systems, the **feholdexcept** function can be used in conjunction with the **feupdateenv** function to write routines that hide spurious floating-point exceptions from their callers.

4 EXAMPLE Hide spurious underflow floating-point exceptions:

```
#include <fenv.h>
double f(double x)
{
      #pragma STDC FENV_ACCESS ON
      double result;
      fenv_t save_env;
      if (feholdexcept(&save_env))
            return /* indication of an environmental problem */;
      // compute result
      if (/* test spurious underflow */)
            if (feclearexcept(FE_UNDERFLOW))
                  return /* indication of an environmental problem */;
      if (feupdateenv(&save_env))
            return /* indication of an environmental problem */;
      return result;
}
```

7.7 Characteristics of floating types `<float.h>`

1 The header **`<float.h>`** defines several macros that expand to various limits and parameters of the standard floating-point types.

2 The macros, their meanings, and the constraints (or restrictions) on their values are listed in 5.2.4.2.2.

7.8 Format conversion of integer types **<inttypes.h>**

1 The header **<inttypes.h>** includes the header **<stdint.h>** and extends it with additional facilities provided by hosted implementations.

2 It declares functions for manipulating greatest-width integers and converting numeric character strings to greatest-width integers, and it declares the type

imaxdiv_t

which is a structure type that is the type of the value returned by the **imaxdiv** function. For each type declared in **<stdint.h>**, it defines corresponding macros for conversion specifiers for use with the formatted input/output functions.[219)]

Forward references: integer types **<stdint.h>** (7.20), formatted input/output functions (7.21.6), formatted wide character input/output functions (7.29.2).

7.8.1 Macros for format specifiers

1 Each of the following object-like macros expands to a character string literal containing a conversion specifier, possibly modified by a length modifier, suitable for use within the format argument of a formatted input/output function when converting the corresponding integer type. These macro names have the general form of **PRI** (character string literals for the **fprintf** and **fwprintf** family) or **SCN** (character string literals for the **fscanf** and **fwscanf** family),[220)] followed by the conversion specifier, followed by a name corresponding to a similar type name in 7.20.1. In these names, *N* represents the width of the type as described in 7.20.1. For example, **PRIdFAST32** can be used in a format string to print the value of an integer of type **int_fast32_t**.

2 The **fprintf** macros for signed integers are:

PRId*N*	**PRIdLEAST***N*	**PRIdFAST***N*	**PRIdMAX**	**PRIdPTR**
PRIi*N*	**PRIiLEAST***N*	**PRIiFAST***N*	**PRIiMAX**	**PRIiPTR**

3 The **fprintf** macros for unsigned integers are:

PRIo*N*	**PRIoLEAST***N*	**PRIoFAST***N*	**PRIoMAX**	**PRIoPTR**
PRIu*N*	**PRIuLEAST***N*	**PRIuFAST***N*	**PRIuMAX**	**PRIuPTR**
PRIx*N*	**PRIxLEAST***N*	**PRIxFAST***N*	**PRIxMAX**	**PRIxPTR**
PRIX*N*	**PRIXLEAST***N*	**PRIXFAST***N*	**PRIXMAX**	**PRIXPTR**

4 The **fscanf** macros for signed integers are:

---

219) See ‘‘future library directions’’ (7.31.5).

220) Separate macros are given for use with **fprintf** and **fscanf** functions because, in the general case, different format specifiers may be required for **fprintf** and **fscanf**, even when the type is the same.

SCNd*N*	**SCNdLEAST***N*	**SCNdFAST***N*	**SCNdMAX**	**SCNdPTR**
SCNi*N*	**SCNiLEAST***N*	**SCNiFAST***N*	**SCNiMAX**	**SCNiPTR**

5 The **fscanf** macros for unsigned integers are:

SCNo*N*	**SCNoLEAST***N*	**SCNoFAST***N*	**SCNoMAX**	**SCNoPTR**
SCNu*N*	**SCNuLEAST***N*	**SCNuFAST***N*	**SCNuMAX**	**SCNuPTR**
SCNx*N*	**SCNxLEAST***N*	**SCNxFAST***N*	**SCNxMAX**	**SCNxPTR**

6 For each type that the implementation provides in **<stdint.h>**, the corresponding **fprintf** macros shall be defined and the corresponding **fscanf** macros shall be defined unless the implementation does not have a suitable **fscanf** length modifier for the type.

7 EXAMPLE

```
#include <inttypes.h>
#include <wchar.h>
int main(void)
{
      uintmax_t i = UINTMAX_MAX;    // this type always exists
      wprintf(L"The largest integer value is %020"
            PRIxMAX "\n", i);
      return 0;
}
```

7.8.2 Functions for greatest-width integer types

7.8.2.1 The imaxabs function

Synopsis

1
```
#include <inttypes.h>
intmax_t imaxabs(intmax_t j);
```

Description

2 The **imaxabs** function computes the absolute value of an integer **j**. If the result cannot be represented, the behavior is undefined.[221)]

Returns

3 The **imaxabs** function returns the absolute value.

221) The absolute value of the most negative number cannot be represented in two's complement.

7.8.2.2 The `imaxdiv` function

Synopsis

1
```
#include <inttypes.h>
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

Description

2 The **imaxdiv** function computes **numer / denom** and **numer % denom** in a single operation.

Returns

3 The **imaxdiv** function returns a structure of type **imaxdiv_t** comprising both the quotient and the remainder. The structure shall contain (in either order) the members **quot** (the quotient) and **rem** (the remainder), each of which has type **intmax_t**. If either part of the result cannot be represented, the behavior is undefined.

7.8.2.3 The `strtoimax` and `strtoumax` functions

Synopsis

1
```
#include <inttypes.h>
intmax_t strtoimax(const char * restrict nptr,
     char ** restrict endptr, int base);
uintmax_t strtoumax(const char * restrict nptr,
     char ** restrict endptr, int base);
```

Description

2 The **strtoimax** and **strtoumax** functions are equivalent to the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions, except that the initial portion of the string is converted to **intmax_t** and **uintmax_t** representation, respectively.

Returns

3 The **strtoimax** and **strtoumax** functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **INTMAX_MAX**, **INTMAX_MIN**, or **UINTMAX_MAX** is returned (according to the return type and sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

Forward references: the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.22.1.4).

7.8.2.4 The `wcstoimax` and `wcstoumax` functions

Synopsis

1
```
#include <stddef.h>            // for wchar_t
#include <inttypes.h>
intmax_t wcstoimax(const wchar_t * restrict nptr,
     wchar_t ** restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t * restrict nptr,
     wchar_t ** restrict endptr, int base);
```

Description

2 The **wcstoimax** and **wcstoumax** functions are equivalent to the **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** functions except that the initial portion of the wide string is converted to **intmax_t** and **uintmax_t** representation, respectively.

Returns

3 The **wcstoimax** function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **INTMAX_MAX**, **INTMAX_MIN**, or **UINTMAX_MAX** is returned (according to the return type and sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

Forward references: the **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** functions (7.29.4.1.2).

7.9 Alternative spellings **<iso646.h>**

1 The header **<iso646.h>** defines the following eleven macros (on the left) that expand to the corresponding tokens (on the right):

```
and       &&
and_eq    &=
bitand    &
bitor     |
compl     ~
not       !
not_eq    !=
or        ||
or_eq     |=
xor       ^
xor_eq    ^=
```

7.10 Sizes of integer types `<limits.h>`

1 The header **`<limits.h>`** defines several macros that expand to various limits and parameters of the standard integer types.

2 The macros, their meanings, and the constraints (or restrictions) on their values are listed in 5.2.4.2.1.

7.11 Localization `<locale.h>`

1 The header `<locale.h>` declares two functions, one type, and defines several macros.

2 The type is

```
struct lconv
```

which contains members related to the formatting of numeric values. The structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are explained in 7.11.2.1. In the `"C"` locale, the members shall have the values specified in the comments.

```
char *decimal_point;       // "."
char *thousands_sep;       // ""
char *grouping;            // ""
char *mon_decimal_point;   // ""
char *mon_thousands_sep;   // ""
char *mon_grouping;        // ""
char *positive_sign;       // ""
char *negative_sign;       // ""
char *currency_symbol;     // ""
char frac_digits;          // CHAR_MAX
char p_cs_precedes;        // CHAR_MAX
char n_cs_precedes;        // CHAR_MAX
char p_sep_by_space;       // CHAR_MAX
char n_sep_by_space;       // CHAR_MAX
char p_sign_posn;          // CHAR_MAX
char n_sign_posn;          // CHAR_MAX
char *int_curr_symbol;     // ""
char int_frac_digits;      // CHAR_MAX
char int_p_cs_precedes;    // CHAR_MAX
char int_n_cs_precedes;    // CHAR_MAX
char int_p_sep_by_space;   // CHAR_MAX
char int_n_sep_by_space;   // CHAR_MAX
char int_p_sign_posn;      // CHAR_MAX
char int_n_sign_posn;      // CHAR_MAX
```

3 The macros defined are **NULL** (described in 7.19); and

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
```

which expand to integer constant expressions with distinct values, suitable for use as the first argument to the **setlocale** function.[222] Additional macro definitions, beginning with the characters **LC_** and an uppercase letter,[223] may also be specified by the implementation.

7.11.1 Locale control

7.11.1.1 The setlocale function

Synopsis

1

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

Description

2 The **setlocale** function selects the appropriate portion of the program's locale as specified by the **category** and **locale** arguments. The **setlocale** function may be used to change or query the program's entire current locale or portions thereof. The value **LC_ALL** for **category** names the program's entire locale; the other values for **category** name only a portion of the program's locale. **LC_COLLATE** affects the behavior of the **strcoll** and **strxfrm** functions. **LC_CTYPE** affects the behavior of the character handling functions[224] and the multibyte and wide character functions. **LC_MONETARY** affects the monetary formatting information returned by the **localeconv** function. **LC_NUMERIC** affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the nonmonetary formatting information returned by the **localeconv** function. **LC_TIME** affects the behavior of the **strftime** and **wcsftime** functions.

3 A value of **"C"** for **locale** specifies the minimal environment for C translation; a value of **""** for **locale** specifies the locale-specific native environment. Other implementation-defined strings may be passed as the second argument to **setlocale**.

---

222) ISO/IEC 9945−2 specifies locale and charmap formats that may be used to specify locales for C.

223) See ‘‘future library directions’’ (7.31.6).

224) The only functions in 7.4 whose behavior is not affected by the current locale are **isdigit** and **isxdigit**.

4 At program startup, the equivalent of

```
setlocale(LC_ALL, "C");
```

is executed.

5 A call to the **setlocale** function may introduce a data race with other calls to the **setlocale** function or with calls to functions that are affected by the current locale. The implementation shall behave as if no library function calls the **setlocale** function.

Returns

6 If a pointer to a string is given for **locale** and the selection can be honored, the **setlocale** function returns a pointer to the string associated with the specified **category** for the new locale. If the selection cannot be honored, the **setlocale** function returns a null pointer and the program's locale is not changed.

7 A null pointer for **locale** causes the **setlocale** function to return a pointer to the string associated with the **category** for the program's current locale; the program's locale is not changed.[225)]

8 The pointer to string returned by the **setlocale** function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **setlocale** function.

Forward references: formatted input/output functions (7.21.6), multibyte/wide character conversion functions (7.22.7), multibyte/wide string conversion functions (7.22.8), numeric conversion functions (7.22.1), the **strcoll** function (7.24.4.3), the **strftime** function (7.27.3.5), the **strxfrm** function (7.24.4.5).

7.11.2 Numeric formatting convention inquiry

7.11.2.1 The localeconv function

Synopsis

1
```
#include <locale.h>
struct lconv *localeconv(void);
```

Description

2 The **localeconv** function sets the components of an object with type **struct lconv** with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

225) The implementation shall arrange to encode in a string the various categories due to a heterogeneous locale when **category** has the value **LC_ALL**.

3 The members of the structure with type **char *** are pointers to strings, any of which (except **decimal_point**) can point to **""**, to indicate that the value is not available in the current locale or is of zero length. Apart from **grouping** and **mon_grouping**, the strings shall start and end in the initial shift state. The members with type **char** are nonnegative numbers, any of which can be **CHAR_MAX** to indicate that the value is not available in the current locale. The members include the following:

char *decimal_point
The decimal-point character used to format nonmonetary quantities.

char *thousands_sep
The character used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

char *grouping
A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

char *mon_decimal_point
The decimal-point used to format monetary quantities.

char *mon_thousands_sep
The separator for groups of digits before the decimal-point in formatted monetary quantities.

char *mon_grouping
A string whose elements indicate the size of each group of digits in formatted monetary quantities.

char *positive_sign
The string used to indicate a nonnegative-valued formatted monetary quantity.

char *negative_sign
The string used to indicate a negative-valued formatted monetary quantity.

char *currency_symbol
The local currency symbol applicable to the current locale.

char frac_digits
The number of fractional digits (those after the decimal-point) to be displayed in a locally formatted monetary quantity.

char p_cs_precedes
Set to 1 or 0 if the **currency_symbol** respectively precedes or succeeds the value for a nonnegative locally formatted monetary quantity.

`char n_cs_precedes`
Set to 1 or 0 if the **`currency_symbol`** respectively precedes or succeeds the value for a negative locally formatted monetary quantity.

`char p_sep_by_space`
Set to a value indicating the separation of the **`currency_symbol`**, the sign string, and the value for a nonnegative locally formatted monetary quantity.

`char n_sep_by_space`
Set to a value indicating the separation of the **`currency_symbol`**, the sign string, and the value for a negative locally formatted monetary quantity.

`char p_sign_posn`
Set to a value indicating the positioning of the **`positive_sign`** for a nonnegative locally formatted monetary quantity.

`char n_sign_posn`
Set to a value indicating the positioning of the **`negative_sign`** for a negative locally formatted monetary quantity.

`char *int_curr_symbol`
The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

`char int_frac_digits`
The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

`char int_p_cs_precedes`
Set to 1 or 0 if the **`int_curr_symbol`** respectively precedes or succeeds the value for a nonnegative internationally formatted monetary quantity.

`char int_n_cs_precedes`
Set to 1 or 0 if the **`int_curr_symbol`** respectively precedes or succeeds the value for a negative internationally formatted monetary quantity.

`char int_p_sep_by_space`
Set to a value indicating the separation of the **`int_curr_symbol`**, the sign string, and the value for a nonnegative internationally formatted monetary quantity.

char int_n_sep_by_space
Set to a value indicating the separation of the **int_curr_symbol**, the sign string, and the value for a negative internationally formatted monetary quantity.

char int_p_sign_posn
Set to a value indicating the positioning of the **positive_sign** for a nonnegative internationally formatted monetary quantity.

char int_n_sign_posn
Set to a value indicating the positioning of the **negative_sign** for a negative internationally formatted monetary quantity.

4 The elements of **grouping** and **mon_grouping** are interpreted according to the following:

CHAR_MAX No further grouping is to be performed.

0 The previous element is to be repeatedly used for the remainder of the digits.

other The integer value is the number of digits that compose the current group. The next element is examined to determine the size of the next group of digits before the current group.

5 The values of **p_sep_by_space**, **n_sep_by_space**, **int_p_sep_by_space**, and **int_n_sep_by_space** are interpreted according to the following:

0 No space separates the currency symbol and value.

1 If the currency symbol and sign string are adjacent, a space separates them from the value; otherwise, a space separates the currency symbol from the value.

2 If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.

For **int_p_sep_by_space** and **int_n_sep_by_space**, the fourth character of **int_curr_symbol** is used instead of a space.

6 The values of **p_sign_posn**, **n_sign_posn**, **int_p_sign_posn**, and **int_n_sign_posn** are interpreted according to the following:

0 Parentheses surround the quantity and currency symbol.

1 The sign string precedes the quantity and currency symbol.

2 The sign string succeeds the quantity and currency symbol.

3 The sign string immediately precedes the currency symbol.

4 The sign string immediately succeeds the currency symbol.

7 The implementation shall behave as if no library function calls the **`localeconv`** function.

Returns

8 The **`localeconv`** function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the **`localeconv`** function. In addition, calls to the **`setlocale`** function with categories **`LC_ALL`**, **`LC_MONETARY`**, or **`LC_NUMERIC`** may overwrite the contents of the structure.

9 EXAMPLE 1 The following table illustrates rules which may well be used by four countries to format monetary quantities.

	Local format		International format	
Country	Positive	Negative	Positive	Negative
Country1	`1.234,56 mk`	`-1.234,56 mk`	`FIM 1.234,56`	`FIM -1.234,56`
Country2	`L.1.234`	`-L.1.234`	`ITL 1.234`	`-ITL 1.234`
Country3	`ƒ 1.234,56`	`ƒ -1.234,56`	`NLG 1.234,56`	`NLG -1.234,56`
Country4	`SFrs.1,234.56`	`SFrs.1,234.56C`	`CHF 1,234.56`	`CHF 1,234.56C`

10 For these four countries, the respective values for the monetary members of the structure returned by **`localeconv`** could be:

	Country1	Country2	Country3	Country4
`mon_decimal_point`	`","`	`""`	`","`	`"."`
`mon_thousands_sep`	`"."`	`"."`	`"."`	`","`
`mon_grouping`	`"\3"`	`"\3"`	`"\3"`	`"\3"`
`positive_sign`	`""`	`""`	`""`	`""`
`negative_sign`	`"-"`	`"-"`	`"-"`	`"C"`
`currency_symbol`	`"mk"`	`"L."`	`"\u0192"`	`"SFrs."`
`frac_digits`	`2`	`0`	`2`	`2`
`p_cs_precedes`	`0`	`1`	`1`	`1`
`n_cs_precedes`	`0`	`1`	`1`	`1`
`p_sep_by_space`	`1`	`0`	`1`	`0`
`n_sep_by_space`	`1`	`0`	`2`	`0`
`p_sign_posn`	`1`	`1`	`1`	`1`
`n_sign_posn`	`1`	`1`	`4`	`2`
`int_curr_symbol`	`"FIM "`	`"ITL "`	`"NLG "`	`"CHF "`
`int_frac_digits`	`2`	`0`	`2`	`2`
`int_p_cs_precedes`	`1`	`1`	`1`	`1`
`int_n_cs_precedes`	`1`	`1`	`1`	`1`
`int_p_sep_by_space`	`1`	`1`	`1`	`1`
`int_n_sep_by_space`	`2`	`1`	`2`	`1`
`int_p_sign_posn`	`1`	`1`	`1`	`1`
`int_n_sign_posn`	`4`	`1`	`4`	`2`

11 EXAMPLE 2 The following table illustrates how the cs_precedes, sep_by_space, and sign_posn members affect the formatted value.

		p_sep_by_space		
p_cs_precedes	**p_sign_posn**	**0**	**1**	**2**
0	0	(1.25$)	(1.25 $)	(1.25$)
	1	+1.25$	+1.25 $	+ 1.25$
	2	1.25$+	1.25 $+	1.25$ +
	3	1.25+$	1.25 +$	1.25+ $
	4	1.25$+	1.25 $+	1.25$ +
1	0	($1.25)	($ 1.25)	($1.25)
	1	+$1.25	+$ 1.25	+ $1.25
	2	$1.25+	$ 1.25+	$1.25 +
	3	+$1.25	+$ 1.25	+ $1.25
	4	$+1.25	$+ 1.25	$ +1.25

7.12 Mathematics `<math.h>`

1 The header **`<math.h>`** declares two types and many mathematical functions and defines several macros. Most synopses specify a family of functions consisting of a principal function with one or more **`double`** parameters, a **`double`** return value, or both; and other functions with the same name but with **`f`** and **`l`** suffixes, which are corresponding functions with **`float`** and **`long double`** parameters, return values, or both.[226) Integer arithmetic functions and conversion functions are discussed later.

2 The types

```
float_t
double_t
```

are floating types at least as wide as **`float`** and **`double`**, respectively, and such that **`double_t`** is at least as wide as **`float_t`**. If **`FLT_EVAL_METHOD`** equals 0, **`float_t`** and **`double_t`** are **`float`** and **`double`**, respectively; if **`FLT_EVAL_METHOD`** equals 1, they are both **`double`**; if **`FLT_EVAL_METHOD`** equals 2, they are both **`long double`**; and for other values of **`FLT_EVAL_METHOD`**, they are otherwise implementation-defined.[227)

3 The macro

```
HUGE_VAL
```

expands to a positive **`double`** constant expression, not necessarily representable as a **`float`**. The macros

```
HUGE_VALF
HUGE_VALL
```

are respectively **`float`** and **`long double`** analogs of **`HUGE_VAL`**.[228)

4 The macro

```
INFINITY
```

expands to a constant expression of type **`float`** representing positive or unsigned infinity, if available; else to a positive constant of type **`float`** that overflows at

226) Particularly on systems with wide expression evaluation, a **`<math.h>`** function might pass arguments and return values in wider format than the synopsis prototype indicates.

227) The types **`float_t`** and **`double_t`** are intended to be the implementation's most efficient types at least as wide as **`float`** and **`double`**, respectively. For **`FLT_EVAL_METHOD`** equal 0, 1, or 2, the type **`float_t`** is the narrowest type used by the implementation to evaluate floating expressions.

228) **`HUGE_VAL`**, **`HUGE_VALF`**, and **`HUGE_VALL`** can be positive infinities in an implementation that supports infinities.

translation time.[229)]

5 The macro

`NAN`

is defined if and only if the implementation supports quiet NaNs for the **`float`** type. It expands to a constant expression of type **`float`** representing a quiet NaN.

6 The *number classification macros*

```
FP_INFINITE
FP_NAN
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
```

represent the mutually exclusive kinds of floating-point values. They expand to integer constant expressions with distinct values. Additional implementation-defined floating-point classifications, with macro definitions beginning with **`FP_`** and an uppercase letter, may also be specified by the implementation.

7 The macro

`FP_FAST_FMA`

is optionally defined. If defined, it indicates that the **`fma`** function generally executes about as fast as, or faster than, a multiply and an add of **`double`** operands.[230)] The macros

```
FP_FAST_FMAF
FP_FAST_FMAL
```

are, respectively, **`float`** and **`long double`** analogs of **`FP_FAST_FMA`**. If defined, these macros expand to the integer constant **`1`**.

8 The macros

```
FP_ILOGB0
FP_ILOGBNAN
```

expand to integer constant expressions whose values are returned by **`ilogb(x)`** if **`x`** is zero or NaN, respectively. The value of **`FP_ILOGB0`** shall be either **`INT_MIN`** or **`-INT_MAX`**. The value of **`FP_ILOGBNAN`** shall be either **`INT_MAX`** or **`INT_MIN`**.

229) In this case, using **`INFINITY`** will violate the constraint in 6.4.4 and thus require a diagnostic.

230) Typically, the **`FP_FAST_FMA`** macro is defined if and only if the **`fma`** function is implemented directly with a hardware multiply-add instruction. Software implementations are expected to be substantially slower.

9 The macros

```
MATH_ERRNO
MATH_ERREXCEPT
```

expand to the integer constants **`1`** and **`2`**, respectively; the macro

```
math_errhandling
```

expands to an expression that has type **`int`** and the value **`MATH_ERRNO`**, **`MATH_ERREXCEPT`**, or the bitwise OR of both. The value of **`math_errhandling`** is constant for the duration of the program. It is unspecified whether **`math_errhandling`** is a macro or an identifier with external linkage. If a macro definition is suppressed or a program defines an identifier with the name **`math_errhandling`**, the behavior is undefined. If the expression **`math_errhandling & MATH_ERREXCEPT`** can be nonzero, the implementation shall define the macros **`FE_DIVBYZERO`**, **`FE_INVALID`**, and **`FE_OVERFLOW`** in **`<fenv.h>`**.

7.12.1 Treatment of error conditions

1 The behavior of each of the functions in **`<math.h>`** is specified for all representable values of its input arguments, except where stated otherwise. Each function shall execute as if it were a single operation without raising **`SIGFPE`** and without generating any of the floating-point exceptions ‘‘invalid’’, ‘‘divide-by-zero’’, or ‘‘overflow’’ except to reflect the result of the function.

2 For all functions, a *domain error* occurs if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any required domain errors; an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function.[231] On a domain error, the function returns an implementation-defined value; if the integer expression **`math_errhandling & MATH_ERRNO`** is nonzero, the integer expression **`errno`** acquires the value **`EDOM`**; if the integer expression **`math_errhandling & MATH_ERREXCEPT`** is nonzero, the ‘‘invalid’’ floating-point exception is raised.

3 Similarly, a *pole error* (also known as a singularity or infinitary) occurs if the mathematical function has an exact infinite result as the finite input argument(s) are approached in the limit (for example, **`log(0.0)`**). The description of each function lists any required pole errors; an implementation may define additional pole errors, provided that such errors are consistent with the mathematical definition of the function. On a pole error, the function returns an implementation-defined value; if the integer expression

231) In an implementation that supports infinities, this allows an infinity as an argument to be a domain error if the mathematical domain of the function does not include the infinity.

math_errhandling & MATH_ERRNO is nonzero, the integer expression **errno** acquires the value **ERANGE**; if the integer expression **math_errhandling & MATH_ERREXCEPT** is nonzero, the ''divide-by-zero'' floating-point exception is raised.

4 Likewise, a *range error* occurs if the mathematical result of the function cannot be represented in an object of the specified type, due to extreme magnitude.

5 A floating result overflows if the magnitude of the mathematical result is finite but so large that the mathematical result cannot be represented without extraordinary roundoff error in an object of the specified type. If a floating result overflows and default rounding is in effect, then the function returns the value of the macro **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** according to the return type, with the same sign as the correct value of the function; if the integer expression **math_errhandling & MATH_ERRNO** is nonzero, the integer expression **errno** acquires the value **ERANGE**; if the integer expression **math_errhandling & MATH_ERREXCEPT** is nonzero, the ''overflow'' floating-point exception is raised.

6 The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type.[232] If the result underflows, the function returns an implementation-defined value whose magnitude is no greater than the smallest normalized positive number in the specified type; if the integer expression **math_errhandling & MATH_ERRNO** is nonzero, whether **errno** acquires the value **ERANGE** is implementation-defined; if the integer expression **math_errhandling & MATH_ERREXCEPT** is nonzero, whether the ''underflow'' floating-point exception is raised is implementation-defined.

7 If a domain, pole, or range error occurs and the integer expression **math_errhandling & MATH_ERRNO** is zero,[233] then **errno** shall either be set to the value corresponding to the error or left unmodified. If no such error occurs, **errno** shall be left unmodified regardless of the setting of **math_errhandling**.

232) The term underflow here is intended to encompass both ''gradual underflow'' as in IEC 60559 and also ''flush-to-zero'' underflow.

233) Math errors are being indicated by the floating-point exception flags rather than by **errno**.

7.12.2 The FP_CONTRACT pragma

Synopsis

1
```
#include <math.h>
#pragma STDC FP_CONTRACT on-off-switch
```

Description

2 The **FP_CONTRACT** pragma can be used to allow (if the state is ‘‘on’’) or disallow (if the state is ‘‘off’’) the implementation to contract expressions (6.5). Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another **FP_CONTRACT** pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another **FP_CONTRACT** pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state (‘‘on’’ or ‘‘off’’) for the pragma is implementation-defined.

7.12.3 Classification macros

1 In the synopses in this subclause, *real-floating* indicates that the argument shall be an expression of real floating type.

7.12.3.1 The fpclassify macro

Synopsis

1
```
#include <math.h>
int fpclassify(real-floating x);
```

Description

2 The **fpclassify** macro classifies its argument value as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.[234)]

Returns

3 The **fpclassify** macro returns the value of the number classification macro appropriate to the value of its argument.

---

234) Since an expression can be evaluated with more range and precision than its type has, it is important to know the type that classification is based on. For example, a normal **long double** value might become subnormal when converted to **double**, and zero when converted to **float**.

7.12.3.2 The `isfinite` macro

Synopsis

1
```
#include <math.h>
int isfinite(real-floating x);
```

Description

2 The **`isfinite`** macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

3 The **`isfinite`** macro returns a nonzero value if and only if its argument has a finite value.

7.12.3.3 The `isinf` macro

Synopsis

1
```
#include <math.h>
int isinf(real-floating x);
```

Description

2 The **`isinf`** macro determines whether its argument value is an infinity (positive or negative). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

3 The **`isinf`** macro returns a nonzero value if and only if its argument has an infinite value.

7.12.3.4 The `isnan` macro

Synopsis

1
```
#include <math.h>
int isnan(real-floating x);
```

Description

2 The **`isnan`** macro determines whether its argument value is a NaN. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.[235)]

---

235) For the **`isnan`** macro, the type for determination does not matter unless the implementation supports NaNs in the evaluation type but not in the semantic type.

Returns

3 The **`isnan`** macro returns a nonzero value if and only if its argument has a NaN value.

7.12.3.5 The `isnormal` macro

Synopsis

1
```
#include <math.h>
int isnormal(real-floating x);
```

Description

2 The **`isnormal`** macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

3 The **`isnormal`** macro returns a nonzero value if and only if its argument has a normal value.

7.12.3.6 The `signbit` macro

Synopsis

1
```
#include <math.h>
int signbit(real-floating x);
```

Description

2 The **`signbit`** macro determines whether the sign of its argument value is negative.[236)]

Returns

3 The **`signbit`** macro returns a nonzero value if and only if the sign of its argument value is negative.

236) The **`signbit`** macro reports the sign of all values, including infinities, zeros, and NaNs. If zero is unsigned, it is treated as positive.

7.12.4 Trigonometric functions

7.12.4.1 The `acos` functions

Synopsis

1
```
#include <math.h>
double acos(double x);
float acosf(float x);
long double acosl(long double x);
```

Description

2 The **`acos`** functions compute the principal value of the arc cosine of **`x`**. A domain error occurs for arguments not in the interval $[-1, +1]$.

Returns

3 The **`acos`** functions return arccos **`x`** in the interval $[0, \pi]$ radians.

7.12.4.2 The `asin` functions

Synopsis

1
```
#include <math.h>
double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

Description

2 The **`asin`** functions compute the principal value of the arc sine of **`x`**. A domain error occurs for arguments not in the interval $[-1, +1]$.

Returns

3 The **`asin`** functions return arcsin **`x`** in the interval $[-\pi/2, +\pi/2]$ radians.

7.12.4.3 The `atan` functions

Synopsis

1
```
#include <math.h>
double atan(double x);
float atanf(float x);
long double atanl(long double x);
```

Description

2 The **`atan`** functions compute the principal value of the arc tangent of **`x`**.

Returns

3 The **atan** functions return arctan **x** in the interval $[-\pi/2, +\pi/2]$ radians.

7.12.4.4 The atan2 functions

Synopsis

1
```
#include <math.h>
double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
```

Description

2 The **atan2** functions compute the value of the arc tangent of **y**/**x**, using the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero.

Returns

3 The **atan2** functions return arctan **y**/**x** in the interval $[-\pi, +\pi]$ radians.

7.12.4.5 The cos functions

Synopsis

1
```
#include <math.h>
double cos(double x);
float cosf(float x);
long double cosl(long double x);
```

Description

2 The **cos** functions compute the cosine of **x** (measured in radians).

Returns

3 The **cos** functions return cos **x**.

7.12.4.6 The sin functions

Synopsis

1
```
#include <math.h>
double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

Description

2 The **sin** functions compute the sine of **x** (measured in radians).

Returns

3 The **sin** functions return $\sin$ **x**.

7.12.4.7 The tan functions

Synopsis

1
```
#include <math.h>
double tan(double x);
float tanf(float x);
long double tanl(long double x);
```

Description

2 The **tan** functions return the tangent of **x** (measured in radians).

Returns

3 The **tan** functions return $\tan$ **x**.

7.12.5 Hyperbolic functions

7.12.5.1 The acosh functions

Synopsis

1
```
#include <math.h>
double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
```

Description

2 The **acosh** functions compute the (nonnegative) arc hyperbolic cosine of **x**. A domain error occurs for arguments less than 1.

Returns

3 The **acosh** functions return arcosh **x** in the interval $[0, +\infty]$.

7.12.5.2 The asinh functions

Synopsis

1
```
#include <math.h>
double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
```

Description

2 The **asinh** functions compute the arc hyperbolic sine of **x**.

Returns

3 The `asinh` functions return arsinh `x`.

7.12.5.3 The `atanh` functions

Synopsis

1
```
#include <math.h>
double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
```

Description

2 The `atanh` functions compute the arc hyperbolic tangent of `x`. A domain error occurs for arguments not in the interval $[-1, +1]$. A pole error may occur if the argument equals -1 or $+1$.

Returns

3 The `atanh` functions return artanh `x`.

7.12.5.4 The `cosh` functions

Synopsis

1
```
#include <math.h>
double cosh(double x);
float coshf(float x);
long double coshl(long double x);
```

Description

2 The `cosh` functions compute the hyperbolic cosine of `x`. A range error occurs if the magnitude of `x` is too large.

Returns

3 The `cosh` functions return cosh `x`.

7.12.5.5 The `sinh` functions

Synopsis

1
```
#include <math.h>
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
```

Description

2 The `sinh` functions compute the hyperbolic sine of `x`. A range error occurs if the magnitude of `x` is too large.

Returns

3 The **sinh** functions return sinh **x**.

7.12.5.6 The tanh functions

Synopsis

1
```
#include <math.h>
double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
```

Description

2 The **tanh** functions compute the hyperbolic tangent of **x**.

Returns

3 The **tanh** functions return tanh **x**.

7.12.6 Exponential and logarithmic functions

7.12.6.1 The exp functions

Synopsis

1
```
#include <math.h>
double exp(double x);
float expf(float x);
long double expl(long double x);
```

Description

2 The **exp** functions compute the base-*e* exponential of **x**. A range error occurs if the magnitude of **x** is too large.

Returns

3 The **exp** functions return $e^{\mathbf{x}}$.

7.12.6.2 The exp2 functions

Synopsis

1
```
#include <math.h>
double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
```

Description

2 The **exp2** functions compute the base-2 exponential of **x**. A range error occurs if the magnitude of **x** is too large.

Returns

3 The **`exp2`** functions return $2^{\mathbf{x}}$.

7.12.6.3 The `expm1` functions

Synopsis

1
```
#include <math.h>
double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
```

Description

2 The **`expm1`** functions compute the base-*e* exponential of the argument, minus 1. A range error occurs if **`x`** is too large.[237)]

Returns

3 The **`expm1`** functions return $e^{\mathbf{x}} - 1$.

7.12.6.4 The `frexp` functions

Synopsis

1
```
#include <math.h>
double frexp(double value, int *exp);
float frexpf(float value, int *exp);
long double frexpl(long double value, int *exp);
```

Description

2 The **`frexp`** functions break a floating-point number into a normalized fraction and an integral power of 2. They store the integer in the **`int`** object pointed to by **`exp`**.

Returns

3 If **`value`** is not a floating-point number or if the integral power of 2 is outside the range of **`int`**, the results are unspecified. Otherwise, the **`frexp`** functions return the value **`x`**, such that **`x`** has a magnitude in the interval $[1/2, 1)$ or zero, and **`value`** equals $\mathbf{x} \times 2^{\mathbf{*exp}}$. If **`value`** is zero, both parts of the result are zero.

237) For small magnitude **`x`**, **`expm1(x)`** is expected to be more accurate than **`exp(x) - 1`**.

7.12.6.5 The `ilogb` functions

Synopsis

1
```
#include <math.h>
int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
```

Description

2 The **ilogb** functions extract the exponent of **x** as a signed **int** value. If **x** is zero they compute the value **FP_ILOGB0**; if **x** is infinite they compute the value **INT_MAX**; if **x** is a NaN they compute the value **FP_ILOGBNAN**; otherwise, they are equivalent to calling the corresponding **logb** function and casting the returned value to type **int**. A domain error or range error may occur if **x** is zero, infinite, or NaN. If the correct value is outside the range of the return type, the numeric result is unspecified.

Returns

3 The **ilogb** functions return the exponent of **x** as a signed **int** value.

Forward references: the **logb** functions (7.12.6.11).

7.12.6.6 The `ldexp` functions

Synopsis

1
```
#include <math.h>
double ldexp(double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
```

Description

2 The **ldexp** functions multiply a floating-point number by an integral power of 2. A range error may occur.

Returns

3 The **ldexp** functions return $\mathbf{x} \times 2^{\mathbf{exp}}$.

7.12.6.7 The `log` functions

Synopsis

1
```
#include <math.h>
double log(double x);
float logf(float x);
long double logl(long double x);
```

Description

2 The **`log`** functions compute the base-e (natural) logarithm of **`x`**. A domain error occurs if the argument is negative. A pole error may occur if the argument is zero.

Returns

3 The **`log`** functions return $\log_e$ **`x`**.

7.12.6.8 The `log10` functions

Synopsis

1
```
#include <math.h>
double log10(double x);
float log10f(float x);
long double log10l(long double x);
```

Description

2 The **`log10`** functions compute the base-10 (common) logarithm of **`x`**. A domain error occurs if the argument is negative. A pole error may occur if the argument is zero.

Returns

3 The **`log10`** functions return $\log_{10}$ **`x`**.

7.12.6.9 The `log1p` functions

Synopsis

1
```
#include <math.h>
double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
```

Description

2 The **`log1p`** functions compute the base-e (natural) logarithm of 1 plus the argument.[238)] A domain error occurs if the argument is less than −1. A pole error may occur if the argument equals −1.

Returns

3 The **`log1p`** functions return $\log_e(1 + $ **`x`**$)$.

238) For small magnitude **`x`**, **`log1p(x)`** is expected to be more accurate than **`log(1 + x)`**.

7.12.6.10 The `log2` functions

Synopsis

1
```
#include <math.h>
double log2(double x);
float log2f(float x);
long double log2l(long double x);
```

Description

2 The **`log2`** functions compute the base-2 logarithm of **`x`**. A domain error occurs if the argument is less than zero. A pole error may occur if the argument is zero.

Returns

3 The **`log2`** functions return $\log_2 \mathbf{x}$.

7.12.6.11 The `logb` functions

Synopsis

1
```
#include <math.h>
double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

Description

2 The **`logb`** functions extract the exponent of **`x`**, as a signed integer value in floating-point format. If **`x`** is subnormal it is treated as though it were normalized; thus, for positive finite **`x`**,

$$1 \le \mathtt{x} \times \mathtt{FLT\_RADIX}^{-\mathtt{logb(x)}} < \mathtt{FLT\_RADIX}$$

A domain error or pole error may occur if the argument is zero.

Returns

3 The **`logb`** functions return the signed exponent of **`x`**.

7.12.6.12 The `modf` functions

Synopsis

1
```
#include <math.h>
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

Description

2 The **`modf`** functions break the argument **`value`** into integral and fractional parts, each of which has the same type and sign as the argument. They store the integral part (in

floating-point format) in the object pointed to by **iptr**.

Returns

3 The **modf** functions return the signed fractional part of **value**.

7.12.6.13 The scalbn and scalbln functions

Synopsis

1
```
#include <math.h>
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
double scalbln(double x, long int n);
float scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);
```

Description

2 The **scalbn** and **scalbln** functions compute $\mathbf{x} \times \mathbf{FLT\_RADIX}^{n}$ efficiently, not normally by computing $\mathbf{FLT\_RADIX}^{n}$ explicitly. A range error may occur.

Returns

3 The **scalbn** and **scalbln** functions return $\mathbf{x} \times \mathbf{FLT\_RADIX}^{n}$.

7.12.7 Power and absolute-value functions

7.12.7.1 The cbrt functions

Synopsis

1
```
#include <math.h>
double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);
```

Description

2 The **cbrt** functions compute the real cube root of **x**.

Returns

3 The **cbrt** functions return $\mathbf{x}^{1/3}$.

7.12.7.2 The `fabs` functions

Synopsis

1
```
#include <math.h>
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
```

Description

2 The **fabs** functions compute the absolute value of a floating-point number **x**.

Returns

3 The **fabs** functions return $|\mathbf{x}|$.

7.12.7.3 The `hypot` functions

Synopsis

1
```
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

Description

2 The **hypot** functions compute the square root of the sum of the squares of **x** and **y**, without undue overflow or underflow. A range error may occur.

3 **Returns**

4 The **hypot** functions return $\sqrt{\mathbf{x}^2 + \mathbf{y}^2}$.

7.12.7.4 The `pow` functions

Synopsis

1
```
#include <math.h>
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
```

Description

2 The **pow** functions compute **x** raised to the power **y**. A domain error occurs if **x** is finite and negative and **y** is finite and not an integer value. A range error may occur. A domain error may occur if **x** is zero and **y** is zero. A domain error or pole error may occur if **x** is zero and **y** is less than zero.

Returns

3 The **pow** functions return $\mathbf{x}^{\mathbf{y}}$.

7.12.7.5 The sqrt functions

Synopsis

1
```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

Description

2 The **sqrt** functions compute the nonnegative square root of **x**. A domain error occurs if the argument is less than zero.

Returns

3 The **sqrt** functions return $\sqrt{\mathbf{x}}$.

7.12.8 Error and gamma functions

7.12.8.1 The erf functions

Synopsis

1
```
#include <math.h>
double erf(double x);
float erff(float x);
long double erfl(long double x);
```

Description

2 The **erf** functions compute the error function of **x**.

Returns

3 The **erf** functions return $\operatorname{erf} \mathbf{x} = \frac{2}{\sqrt{\pi}} \int_0^{\mathbf{x}} e^{-t^2}\, dt$.

7.12.8.2 The erfc functions

Synopsis

1
```
#include <math.h>
double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);
```

Description

2 The **erfc** functions compute the complementary error function of **x**. A range error occurs if **x** is too large.

Returns

3 The **erfc** functions return erfc $\mathbf{x} = 1 - \operatorname{erf} \mathbf{x} = \frac{2}{\sqrt{\pi}} \int_{\mathbf{x}}^{\infty} e^{-t^2} \, dt$.

7.12.8.3 The lgamma functions

Synopsis

1
```
#include <math.h>
double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
```

Description

2 The **lgamma** functions compute the natural logarithm of the absolute value of gamma of **x**. A range error occurs if **x** is too large. A pole error may occur if **x** is a negative integer or zero.

Returns

3 The **lgamma** functions return $\log_e |\Gamma(\mathbf{x})|$.

7.12.8.4 The tgamma functions

Synopsis

1
```
#include <math.h>
double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
```

Description

2 The **tgamma** functions compute the gamma function of **x**. A domain error or pole error may occur if **x** is a negative integer or zero. A range error occurs if the magnitude of **x** is too large and may occur if the magnitude of **x** is too small.

Returns

3 The **tgamma** functions return $\Gamma(\mathbf{x})$.

7.12.9 Nearest integer functions

7.12.9.1 The `ceil` functions

Synopsis

1
```
#include <math.h>
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
```

Description

2 The **`ceil`** functions compute the smallest integer value not less than **`x`**.

Returns

3 The **`ceil`** functions return $\lceil \mathbf{x} \rceil$, expressed as a floating-point number.

7.12.9.2 The `floor` functions

Synopsis

1
```
#include <math.h>
double floor(double x);
float floorf(float x);
long double floorl(long double x);
```

Description

2 The **`floor`** functions compute the largest integer value not greater than **`x`**.

Returns

3 The **`floor`** functions return $\lfloor \mathbf{x} \rfloor$, expressed as a floating-point number.

7.12.9.3 The `nearbyint` functions

Synopsis

1
```
#include <math.h>
double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
```

Description

2 The **`nearbyint`** functions round their argument to an integer value in floating-point format, using the current rounding direction and without raising the ‘‘inexact’’ floating-point exception.

Returns

3 The **nearbyint** functions return the rounded integer value.

7.12.9.4 The rint functions

Synopsis

1
```
#include <math.h>
double rint(double x);
float rintf(float x);
long double rintl(long double x);
```

Description

2 The **rint** functions differ from the **nearbyint** functions (7.12.9.3) only in that the **rint** functions may raise the ‘‘inexact’’ floating-point exception if the result differs in value from the argument.

Returns

3 The **rint** functions return the rounded integer value.

7.12.9.5 The lrint and llrint functions

Synopsis

1
```
#include <math.h>
long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);
long long int llrint(double x);
long long int llrintf(float x);
long long int llrintl(long double x);
```

Description

2 The **lrint** and **llrint** functions round their argument to the nearest integer value, rounding according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns

3 The **lrint** and **llrint** functions return the rounded integer value.

7.12.9.6 The `round` functions

Synopsis

1
```
#include <math.h>
double round(double x);
float roundf(float x);
long double roundl(long double x);
```

Description

2 The **`round`** functions round their argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

Returns

3 The **`round`** functions return the rounded integer value.

7.12.9.7 The `lround` and `llround` functions

Synopsis

1
```
#include <math.h>
long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
```

Description

2 The **`lround`** and **`llround`** functions round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns

3 The **`lround`** and **`llround`** functions return the rounded integer value.

7.12.9.8 The `trunc` functions

Synopsis

1
```
#include <math.h>
double trunc(double x);
float truncf(float x);
long double truncl(long double x);
```

Description

2 The **`trunc`** functions round their argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument.

Returns

3 The **`trunc`** functions return the truncated integer value.

7.12.10 Remainder functions

7.12.10.1 The `fmod` functions

Synopsis

1
```
#include <math.h>
double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
```

Description

2 The **`fmod`** functions compute the floating-point remainder of **`x`**/**`y`**.

Returns

3 The **`fmod`** functions return the value **`x`** – *n***`y`**, for some integer *n* such that, if **`y`** is nonzero, the result has the same sign as **`x`** and magnitude less than the magnitude of **`y`**. If **`y`** is zero, whether a domain error occurs or the **`fmod`** functions return zero is implementation-defined.

7.12.10.2 The `remainder` functions

Synopsis

1
```
#include <math.h>
double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
```

Description

2 The **`remainder`** functions compute the remainder **`x`** REM **`y`** required by IEC 60559.[239)]

239) "When $y \neq 0$, the remainder $r = x$ REM y is defined regardless of the rounding mode by the mathematical relation $r = x - ny$, where n is the integer nearest the exact value of x/y; whenever $|\,n - x/y\,| = 1/2$, then n is even. If $r = 0$, its sign shall be that of x." This definition is applicable for all implementations.

Returns

3 The **remainder** functions return **x** REM **y**. If **y** is zero, whether a domain error occurs or the functions return zero is implementation defined.

7.12.10.3 The remquo functions

Synopsis

1
```
#include <math.h>
double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y,
     int *quo);
```

Description

2 The **remquo** functions compute the same remainder as the **remainder** functions. In the object pointed to by **quo** they store a value whose sign is the sign of **x**/**y** and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of **x**/**y**, where n is an implementation-defined integer greater than or equal to 3.

Returns

3 The **remquo** functions return **x** REM **y**. If **y** is zero, the value stored in the object pointed to by **quo** is unspecified and whether a domain error occurs or the functions return zero is implementation defined.

7.12.11 Manipulation functions

7.12.11.1 The copysign functions

Synopsis

1
```
#include <math.h>
double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
```

Description

2 The **copysign** functions produce a value with the magnitude of **x** and the sign of **y**. They produce a NaN (with the sign of **y**) if **x** is a NaN. On implementations that represent a signed zero but do not treat negative zero consistently in arithmetic operations, the **copysign** functions regard the sign of zero as positive.

Returns

3 The **copysign** functions return a value with the magnitude of **x** and the sign of **y**.

7.12.11.2 The `nan` functions

Synopsis

1
```
#include <math.h>
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

Description

2 The call nan("*n-char-sequence*") is equivalent to strtod("NAN(*n-char-sequence*)", (char**) NULL); the call `nan("")` is equivalent to `strtod("NAN()", (char**) NULL)`. If `tagp` does not point to an n-char sequence or an empty string, the call is equivalent to `strtod("NAN", (char**) NULL)`. Calls to `nanf` and `nanl` are equivalent to the corresponding calls to `strtof` and `strtold`.

Returns

3 The `nan` functions return a quiet NaN, if available, with content indicated through `tagp`. If the implementation does not support quiet NaNs, the functions return zero.

Forward references: the `strtod`, `strtof`, and `strtold` functions (7.22.1.3).

7.12.11.3 The `nextafter` functions

Synopsis

1
```
#include <math.h>
double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
```

Description

2 The `nextafter` functions determine the next representable value, in the type of the function, after `x` in the direction of `y`, where `x` and `y` are first converted to the type of the function.[240] The `nextafter` functions return `y` if `x` equals `y`. A range error may occur if the magnitude of x is the largest finite value representable in the type and the result is infinite or not representable in the type.

Returns

3 The `nextafter` functions return the next representable value in the specified format after `x` in the direction of `y`.

240) The argument values are converted to the type of the function, even by a macro implementation of the function.

7.12.11.4 The `nexttoward` functions

Synopsis

1
```
#include <math.h>
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

Description

2 The **`nexttoward`** functions are equivalent to the **`nextafter`** functions except that the second parameter has type **`long double`** and the functions return **`y`** converted to the type of the function if **`x`** equals **`y`**.[241)]

7.12.12 Maximum, minimum, and positive difference functions

7.12.12.1 The `fdim` functions

Synopsis

1
```
#include <math.h>
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
```

Description

2 The **`fdim`** functions determine the *positive difference* between their arguments:

$$\begin{cases} \mathbf{x} - \mathbf{y} & \text{if } \mathbf{x} > \mathbf{y} \\ +0 & \text{if } \mathbf{x} \le \mathbf{y} \end{cases}$$

A range error may occur.

Returns

3 The **`fdim`** functions return the positive difference value.

7.12.12.2 The `fmax` functions

Synopsis

1
```
#include <math.h>
double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
```

241) The result of the **`nexttoward`** functions is determined in the type of the function, without loss of range or precision in a floating second argument.

Description

2 The **fmax** functions determine the maximum numeric value of their arguments.[242)]

Returns

3 The **fmax** functions return the maximum numeric value of their arguments.

7.12.12.3 The fmin functions

Synopsis

1
```
#include <math.h>
double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

Description

2 The **fmin** functions determine the minimum numeric value of their arguments.[243)]

Returns

3 The **fmin** functions return the minimum numeric value of their arguments.

7.12.13 Floating multiply-add

7.12.13.1 The fma functions

Synopsis

1
```
#include <math.h>
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y,
     long double z);
```

Description

2 The **fma** functions compute $(\mathbf{x} \times \mathbf{y}) + \mathbf{z}$, rounded as one ternary operation: they compute the value (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur.

Returns

3 The **fma** functions return $(\mathbf{x} \times \mathbf{y}) + \mathbf{z}$, rounded as one ternary operation.

---

242) NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, then the **fmax** functions choose the numeric value. See F.10.9.2.

243) The **fmin** functions are analogous to the **fmax** functions in their treatment of NaNs.

7.12.14 Comparison macros

1 The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values exactly one of the relationships — *less*, *greater*, and *equal* — is true. Relational operators may raise the ‘‘invalid’’ floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the *unordered* relationship is true.[244] The following subclauses provide macros that are *quiet* (non floating-point exception raising) versions of the relational operators, and other comparison macros that facilitate writing efficient code that accounts for NaNs without suffering the ‘‘invalid’’ floating-point exception. In the synopses in this subclause, *real-floating* indicates that the argument shall be an expression of real floating type[245] (both arguments need not have the same type).[246]

7.12.14.1 The `isgreater` macro

Synopsis

1
```
#include <math.h>
int isgreater(real-floating x, real-floating y);
```

Description

2 The **`isgreater`** macro determines whether its first argument is greater than its second argument. The value of **`isgreater(x, y)`** is always equal to **`(x) > (y)`**; however, unlike **`(x) > (y)`**, **`isgreater(x, y)`** does not raise the ‘‘invalid’’ floating-point exception when **`x`** and **`y`** are unordered.

Returns

3 The **`isgreater`** macro returns the value of **`(x) > (y)`**.

7.12.14.2 The `isgreaterequal` macro

Synopsis

1
```
#include <math.h>
int isgreaterequal(real-floating x, real-floating y);
```

244) IEC 60559 requires that the built-in relational operators raise the ‘‘invalid’’ floating-point exception if the operands compare unordered, as an error indicator for programs written without consideration of NaNs; the result in these cases is false.

245) If any argument is of integer type, or any other type that is not a real floating type, the behavior is undefined.

246) Whether an argument represented in a format wider than its semantic type is converted to the semantic type is unspecified.

Description

2 The `isgreaterequal` macro determines whether its first argument is greater than or equal to its second argument. The value of `isgreaterequal(x, y)` is always equal to `(x) >= (y)`; however, unlike `(x) >= (y)`, `isgreaterequal(x, y)` does not raise the ‘‘invalid’’ floating-point exception when `x` and `y` are unordered.

Returns

3 The `isgreaterequal` macro returns the value of `(x) >= (y)`.

7.12.14.3 The `isless` macro

Synopsis

1
```
#include <math.h>
int isless(real-floating x, real-floating y);
```

Description

2 The `isless` macro determines whether its first argument is less than its second argument. The value of `isless(x, y)` is always equal to `(x) < (y)`; however, unlike `(x) < (y)`, `isless(x, y)` does not raise the ‘‘invalid’’ floating-point exception when `x` and `y` are unordered.

Returns

3 The `isless` macro returns the value of `(x) < (y)`.

7.12.14.4 The `islessequal` macro

Synopsis

1
```
#include <math.h>
int islessequal(real-floating x, real-floating y);
```

Description

2 The `islessequal` macro determines whether its first argument is less than or equal to its second argument. The value of `islessequal(x, y)` is always equal to `(x) <= (y)`; however, unlike `(x) <= (y)`, `islessequal(x, y)` does not raise the ‘‘invalid’’ floating-point exception when `x` and `y` are unordered.

Returns

3 The `islessequal` macro returns the value of `(x) <= (y)`.

7.12.14.5 The `islessgreater` macro

Synopsis

1
```
#include <math.h>
int islessgreater(real-floating x, real-floating y);
```

Description

2 The **`islessgreater`** macro determines whether its first argument is less than or greater than its second argument. The **`islessgreater(x, y)`** macro is similar to **`(x) < (y) || (x) > (y)`**; however, **`islessgreater(x, y)`** does not raise the ‘‘invalid’’ floating-point exception when **`x`** and **`y`** are unordered (nor does it evaluate **`x`** and **`y`** twice).

Returns

3 The **`islessgreater`** macro returns the value of **`(x) < (y) || (x) > (y)`**.

7.12.14.6 The `isunordered` macro

Synopsis

1
```
#include <math.h>
int isunordered(real-floating x, real-floating y);
```

Description

2 The **`isunordered`** macro determines whether its arguments are unordered.

Returns

3 The **`isunordered`** macro returns 1 if its arguments are unordered and 0 otherwise.

7.13 Nonlocal jumps `<setjmp.h>`

1 The header **`<setjmp.h>`** defines the macro **`setjmp`**, and declares one function and one type, for bypassing the normal function call and return discipline.[247)]

2 The type declared is

```
jmp_buf
```

which is an array type suitable for holding the information needed to restore a calling environment. The environment of a call to the **`setjmp`** macro consists of information sufficient for a call to the **`longjmp`** function to return execution to the correct block and invocation of that block, were it called recursively. It does not include the state of the floating-point status flags, of open files, or of any other component of the abstract machine.

3 It is unspecified whether **`setjmp`** 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 **`setjmp`**, the behavior is undefined.

7.13.1 Save calling environment

7.13.1.1 The `setjmp` macro

Synopsis

1
```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description

2 The **`setjmp`** macro saves its calling environment in its **`jmp_buf`** argument for later use by the **`longjmp`** function.

Returns

3 If the return is from a direct invocation, the **`setjmp`** macro returns the value zero. If the return is from a call to the **`longjmp`** function, the **`setjmp`** macro returns a nonzero value.

Environmental limits

4 An invocation of the **`setjmp`** macro shall appear only in one of the following contexts:

— the entire controlling expression of a selection or iteration statement;

— one operand of a relational or equality operator with the other operand an integer constant expression, with the resulting expression being the entire controlling

247) These functions are useful for dealing with unusual conditions encountered in a low-level function of a program.

expression of a selection or iteration statement;

— the operand of a unary **!** operator with the resulting expression being the entire controlling expression of a selection or iteration statement; or

— the entire expression of an expression statement (possibly cast to **void**).

5 If the invocation appears in any other context, the behavior is undefined.

7.13.2 Restore calling environment

7.13.2.1 The longjmp function

Synopsis

1
```
#include <setjmp.h>
_Noreturn void longjmp(jmp_buf env, int val);
```

Description

2 The **longjmp** function restores the environment saved by the most recent invocation of the **setjmp** macro in the same invocation of the program with the corresponding **jmp_buf** argument. If there has been no such invocation, or if the invocation was from another thread of execution, or if the function containing the invocation of the **setjmp** macro has terminated execution[248] in the interim, or if the invocation of the **setjmp** macro was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined.

3 All accessible objects have values, and all other components of the abstract machine[249] have state, as of the time the **longjmp** function was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding **setjmp** macro that do not have volatile-qualified type and have been changed between the **setjmp** invocation and **longjmp** call are indeterminate.

Returns

4 After **longjmp** is completed, thread execution continues as if the corresponding invocation of the **setjmp** macro had just returned the value specified by **val**. The **longjmp** function cannot cause the **setjmp** macro to return the value 0; if **val** is 0, the **setjmp** macro returns the value 1.

5 EXAMPLE The **longjmp** function that returns control back to the point of the **setjmp** invocation might cause memory associated with a variable length array object to be squandered.

---

248) For example, by executing a **return** statement or because another **longjmp** call has caused a transfer to a **setjmp** invocation in a function earlier in the set of nested calls.

249) This includes, but is not limited to, the floating-point status flags and the state of open files.

```
#include <setjmp.h>
jmp_buf buf;
void g(int n);
void h(int n);
int n = 6;

void f(void)
{
      int x[n];           // valid: f is not terminated
      setjmp(buf);
      g(n);
}

void g(int n)
{
      int a[n];           // a may remain allocated
      h(n);
}

void h(int n)
{
      int b[n];           // b may remain allocated
      longjmp(buf, 2);    // might cause memory loss
}
```

7.14 Signal handling **<signal.h>**

1 The header **<signal.h>** declares a type and two functions and defines several macros, for handling various *signals* (conditions that may be reported during program execution).

2 The type defined is

sig_atomic_t

which is the (possibly volatile-qualified) integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

3 The macros defined are

SIG_DFL
SIG_ERR
SIG_IGN

which expand to constant expressions with distinct values that have type compatible with the second argument to, and the return value of, the **signal** function, and whose values compare unequal to the address of any declarable function; and the following, which expand to positive integer constant expressions with type **int** and distinct values that are the signal numbers, each corresponding to the specified condition:

SIGABRT abnormal termination, such as is initiated by the **abort** function

SIGFPE an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow

SIGILL detection of an invalid function image, such as an invalid instruction

SIGINT receipt of an interactive attention signal

SIGSEGV an invalid access to storage

SIGTERM a termination request sent to the program

4 An implementation need not generate any of these signals, except as a result of explicit calls to the **raise** function. Additional signals and pointers to undeclarable functions, with macro definitions beginning, respectively, with the letters **SIG** and an uppercase letter or with **SIG_** and an uppercase letter,[250)] may also be specified by the implementation. The complete set of signals, their semantics, and their default handling is implementation-defined; all signal numbers shall be positive.

250) See ‘‘future library directions’’ (7.31.7). The names of the signal numbers reflect the following terms (respectively): abort, floating-point exception, illegal instruction, interrupt, segmentation violation, and termination.

7.14.1 Specify signal handling

7.14.1.1 The `signal` function

Synopsis

1
```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Description

2 The **signal** function chooses one of three ways in which receipt of the signal number **sig** is to be subsequently handled. If the value of **func** is **SIG_DFL**, default handling for that signal will occur. If the value of **func** is **SIG_IGN**, the signal will be ignored. Otherwise, **func** shall point to a function to be called when that signal occurs. An invocation of such a function because of a signal, or (recursively) of any further functions called by that invocation (other than functions in the standard library),[251)] is called a *signal handler*.

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, 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.[252)]

---

251) This includes functions called indirectly via standard library functions (e.g., a **SIGABRT** handler called via the **abort** function).

252) If any signal is generated by an asynchronous signal handler, the behavior is undefined.

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);
```

Description

2 The `raise` function carries out the actions described in 7.14.1.1 for the signal `sig`. If a signal handler is called, the `raise` function shall not return until after the signal handler does.

Returns

3 The `raise` function returns zero if successful, nonzero if unsuccessful.

7.15 Alignment `<stdalign.h>`

1 The header `<stdalign.h>` defines four macros.

2 The macro

```
alignas
```

expands to `_Alignas`; the macro

```
alignof
```

expands to `_Alignof`.

3 The remaining macros are suitable for use in `#if` preprocessing directives. They are

```
__alignas_is_defined
```

and

```
__alignof_is_defined
```

which both expand to the integer constant 1.

7.16 Variable arguments `<stdarg.h>`

1 The header **`<stdarg.h>`** declares a type and defines four macros, for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

2 A function may be called with a variable number of arguments of varying types. As described in 6.9.1, its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated *parmN* in this description.

3 The type declared is

```
va_list
```

which is a complete object type suitable for holding information needed by the macros **`va_start`**, **`va_arg`**, **`va_end`**, and **`va_copy`**. If access to the varying arguments is desired, the called function shall declare an object (generally referred to as **`ap`** in this subclause) having type **`va_list`**. The object **`ap`** may be passed as an argument to another function; if that function invokes the **`va_arg`** macro with parameter **`ap`**, the value of **`ap`** in the calling function is indeterminate and shall be passed to the **`va_end`** macro prior to any further reference to **`ap`**.[253)]

7.16.1 Variable argument list access macros

1 The **`va_start`** and **`va_arg`** macros described in this subclause shall be implemented as macros, not functions. It is unspecified whether **`va_copy`** and **`va_end`** are macros or identifiers 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 same name, the behavior is undefined. Each invocation of the **`va_start`** and **`va_copy`** macros shall be matched by a corresponding invocation of the **`va_end`** macro in the same function.

7.16.1.1 The `va_arg` macro

Synopsis

1
```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Description

2 The **`va_arg`** macro expands to an expression that has the specified type and the value of the next argument in the call. The parameter **`ap`** shall have been initialized by the **`va_start`** or **`va_copy`** macro (without an intervening invocation of the **`va_end`**

253) It is permitted to create a pointer to a **`va_list`** and pass that pointer to another function, in which case the original function may make further use of the original list after the other function returns.

macro for the same **ap**). Each invocation of the **va_arg** macro modifies **ap** so that the values of successive arguments are returned in turn. The parameter *type* shall be a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a ***** to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined, except for the following cases:

— one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;

— one type is pointer to **void** and the other is a pointer to a character type.

Returns

3 The first invocation of the **va_arg** macro after that of the **va_start** macro returns the value of the argument after that specified by *parmN*. Successive invocations return the values of the remaining arguments in succession.

7.16.1.2 The va_copy macro

Synopsis

1
```
#include <stdarg.h>
void va_copy(va_list dest, va_list src);
```

Description

2 The **va_copy** macro initializes **dest** as a copy of **src**, as if the **va_start** macro had been applied to **dest** followed by the same sequence of uses of the **va_arg** macro as had previously been used to reach the present state of **src**. Neither the **va_copy** nor **va_start** macro shall be invoked to reinitialize **dest** without an intervening invocation of the **va_end** macro for the same **dest**.

Returns

3 The **va_copy** macro returns no value.

7.16.1.3 The va_end macro

Synopsis

1
```
#include <stdarg.h>
void va_end(va_list ap);
```

Description

2 The **va_end** macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of the **va_start** macro, or the function containing the expansion of the **va_copy** macro, that initialized the **va_list ap**. The **va_end** macro may modify **ap** so that it is no longer usable (without being reinitialized

by the `va_start` or `va_copy` macro). If there is no corresponding invocation of the `va_start` or `va_copy` macro, or if the `va_end` macro is not invoked before the return, the behavior is undefined.

Returns

3 The `va_end` macro returns no value.

7.16.1.4 The `va_start` macro

Synopsis

1

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

Description

2 The `va_start` macro shall be invoked before any access to the unnamed arguments.

3 The `va_start` macro initializes `ap` for subsequent use by the `va_arg` and `va_end` macros. Neither the `va_start` nor `va_copy` macro shall be invoked to reinitialize `ap` without an intervening invocation of the `va_end` macro for the same `ap`.

4 The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the `, ...`). If the parameter *parmN* is declared with the `register` storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

Returns

5 The `va_start` macro returns no value.

6 EXAMPLE 1 The function `f1` gathers into an array a list of arguments that are pointers to strings (but not more than `MAXARGS` arguments), then passes the array as a single argument to function `f2`. The number of pointers is specified by the first argument to `f1`.

```
#include <stdarg.h>
#define MAXARGS   31

void f1(int n_ptrs, ...)
{
      va_list ap;
      char *array[MAXARGS];
      int ptr_no = 0;
```

```
        if (n_ptrs > MAXARGS)
              n_ptrs = MAXARGS;
        va_start(ap, n_ptrs);
        while (ptr_no < n_ptrs)
              array[ptr_no++] = va_arg(ap, char *);
        va_end(ap);
        f2(n_ptrs, array);
  }
```

Each call to **f1** is required to have visible the definition of the function or a declaration such as

```
  void f1(int, ...);
```

7 EXAMPLE 2 The function **f3** is similar, but saves the status of the variable argument list after the indicated number of arguments; after **f2** has been called once with the whole list, the trailing part of the list is gathered again and passed to function **f4**.

```
  #include <stdarg.h>
  #define MAXARGS 31

  void f3(int n_ptrs, int f4_after, ...)
  {
        va_list ap, ap_save;
        char *array[MAXARGS];
        int ptr_no = 0;
        if (n_ptrs > MAXARGS)
              n_ptrs = MAXARGS;
        va_start(ap, f4_after);
        while (ptr_no < n_ptrs) {
              array[ptr_no++] = va_arg(ap, char *);
              if (ptr_no == f4_after)
                    va_copy(ap_save, ap);
        }
        va_end(ap);
        f2(n_ptrs, array);

        // Now process the saved copy.

        n_ptrs -= f4_after;
        ptr_no = 0;
        while (ptr_no < n_ptrs)
              array[ptr_no++] = va_arg(ap_save, char *);
        va_end(ap_save);
        f4(n_ptrs, array);
  }
```

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.[254)]

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 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 one of the atomic types.

— 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`**.

254) See ‘‘future library directions’’ (7.31.8).

— 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 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 objects.

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 using **`ATOMIC_VAR_INIT`** 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.

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.

Returns

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

5 EXAMPLE

```
atomic_int guide;
atomic_init(&guide, 42);
```

7.17.3 Order and consistency

1 The enumerated type **`memory_order`** specifies the detailed regular (non-atomic) memory synchronization operations as defined in 5.1.2.4 and may provide for operation ordering. Its enumeration constants are as follows:[255)]

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

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

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

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

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

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

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

— if *A* exists, the result of some modification of *M* in the visible sequence of side effects with respect to *B* that is not **`memory_order_seq_cst`** and that does not happen before *A*, or

255) See ‘‘future library directions’’ (7.31.8).

— if A does not exist, the result of some modification of M in the visible sequence of side effects with respect to B 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 must still guarantee that any given atomic access to a particular atomic object be 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 operations A and B on an atomic object M, where A modifies M and B takes its value, if 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, then B observes either the effects of A or a later modification of M in its modification order.

12 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.

13 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.[256] The ordering of evaluations in this sequence shall be such that

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

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

14 NOTE 3 The second requirement disallows ''out-of-thin-air'', or ''speculative'' stores of atomics when relaxed atomics are used. Since unordered operations are involved, evaluations may appear in this sequence out of thread order. For example, with **`x`** and **`y`** initially zero,

256) Among other implications, atomic variables shall not decay.

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

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

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

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

On the other hand,

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

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

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

Recommended practice

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

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

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

However, this is not useful behavior, and implementations should not allow it.

16 Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

7.17.3.1 The `kill_dependency` macro

Synopsis

1
```
#include <stdatomic.h>
type kill_dependency(type y);
```

Description

2 The **`kill_dependency`** macro terminates a dependency chain; the argument does not carry a dependency to the return value.

Returns

3 The **`kill_dependency`** macro returns the value of **`y`**.

7.17.4 Fences

1 This subclause introduces synchronization primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A fence with acquire semantics is called an *acquire fence*; a fence with release semantics is called a *release fence*.

2 A release fence *A* synchronizes with an acquire fence *B* if there exist atomic operations *X* and *Y*, both operating on some atomic object *M*, such that *A* is sequenced before *X*, *X* modifies *M*, *Y* is sequenced before *B*, and *Y* reads the value written by *X* or a value written by any side effect in the hypothetical release sequence *X* would head if it were a release operation.

3 A release fence *A* synchronizes with an atomic operation *B* that performs an acquire operation on an atomic object *M* if there exists an atomic operation *X* such that *A* is sequenced before *X*, *X* modifies *M*, and *B* reads the value written by *X* or a value written by any side effect in the hypothetical release sequence *X* would head if it were a release operation.

4 An atomic operation *A* that is a release operation on an atomic object *M* synchronizes with an acquire fence *B* if there exists some atomic operation *X* on *M* such that *X* is sequenced before *B* and reads the value written by *A* or a value written by any side effect in the release sequence headed by *A*.

7.17.4.1 The `atomic_thread_fence` function

Synopsis

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

Description

2 Depending on the value of **`order`**, this operation:

— has no effects, if **`order == memory_order_relaxed`**;

— is an acquire fence, if **`order == memory_order_acquire`** or **`order == memory_order_consume`**;

— is a release fence, if **`order == memory_order_release`**;

— is both an acquire fence and a release fence, if **`order == memory_order_acq_rel`**;

— is a sequentially consistent acquire and release fence, if **`order == memory_order_seq_cst`**.

Returns

3 The **`atomic_thread_fence`** function returns no value.

7.17.4.2 The `atomic_signal_fence` function

Synopsis

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

Description

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

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

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

Returns

5 The **`atomic_signal_fence`** function returns no value.

7.17.5 Lock-free property

1 The atomic lock-free macros indicate the lock-free property of integer and address atomic 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 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 the object pointed to by **obj** is lock-free.

Returns

3 The **atomic_is_lock_free** generic function returns nonzero (true) if and only if the object's operations are lock-free. The result of a lock-free query on one object cannot be inferred from the result of a lock-free query on another object.

7.17.6 Atomic integer types

1 For each line in the following table,[257] the atomic type name is declared as a type that has the same representation and alignment requirements as the corresponding direct type.[258]

257) See ‘‘future library directions’’ (7.31.8).

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

Atomic type name	Direct type
`atomic_bool`	`_Atomic _Bool`
`atomic_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`
`atomic_uint_least16_t`	`_Atomic uint_least16_t`
`atomic_int_least32_t`	`_Atomic int_least32_t`
`atomic_uint_least32_t`	`_Atomic uint_least32_t`
`atomic_int_least64_t`	`_Atomic int_least64_t`
`atomic_uint_least64_t`	`_Atomic uint_least64_t`
`atomic_int_fast8_t`	`_Atomic int_fast8_t`
`atomic_uint_fast8_t`	`_Atomic uint_fast8_t`
`atomic_int_fast16_t`	`_Atomic int_fast16_t`
`atomic_uint_fast16_t`	`_Atomic uint_fast16_t`
`atomic_int_fast32_t`	`_Atomic int_fast32_t`
`atomic_uint_fast32_t`	`_Atomic uint_fast32_t`
`atomic_int_fast64_t`	`_Atomic int_fast64_t`
`atomic_uint_fast64_t`	`_Atomic uint_fast64_t`
`atomic_intptr_t`	`_Atomic intptr_t`
`atomic_uintptr_t`	`_Atomic uintptr_t`
`atomic_size_t`	`_Atomic size_t`
`atomic_ptrdiff_t`	`_Atomic ptrdiff_t`
`atomic_intmax_t`	`_Atomic intmax_t`
`atomic_uintmax_t`	`_Atomic uintmax_t`

2 The semantics of the operations on these types are defined in 7.17.7. *

3 NOTE The representation of atomic integer types need not have the same size as their corresponding regular types. They should have the same size whenever possible, as it eases effort required to port existing code.

7.17.7 Operations on atomic types

1 There are only a few kinds of operations on atomic types, though there are many instances of those kinds. This subclause specifies each general kind.

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(volatile A *object);
C atomic_load_explicit(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

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. Atomically, compares the value pointed to by **object** for equality with that in **expected**, and if true, replaces the value pointed to by **object** with **desired**, and if false, updates the value in **expected** with the value 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).

3 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));
```

4 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.

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

6 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

7 The result of the comparison.

7.17.7.5 The `atomic_fetch` and modify generic functions

1 The following operations perform arithmetic and bitwise computations. All of these operations are applicable to an object of any atomic integer type. None of these operations is applicable to **`atomic_bool`**. 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

2
```
#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

3 Atomically replaces the value pointed to by **`object`** with the result of the computation applied to the value pointed to by **`object`** and the given operand. Memory is affected

according to the value of **order**. These operations are atomic read-modify-write operations (5.1.2.4). For signed integer types, arithmetic is defined to use two's complement representation with silent wrap-around on overflow; there are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

Returns

4 Atomically, the value pointed to by **object** immediately before the effects.

5 NOTE The operation of the **atomic_fetch** and modify generic functions are nearly equivalent to the operation of the corresponding *op*= compound assignment operators. The only differences are that the compound assignment operators are not guaranteed to operate atomically, and the value yielded by a compound assignment operator is the updated value of the object, whereas the value returned by the **atomic_fetch** and modify generic functions is the previous value of the atomic object.

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 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 International standard. 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 sets the value pointed to by **object** to true. Memory is affected according to the value of **order**. These operations are atomic read-modify-write operations (5.1.2.4).

Returns

3 Atomically, the value of the object immediately before the effects.

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 sets the value pointed to by **`object`** to false. 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 header **`<stdbool.h>`** defines four macros.

2 The macro

```
bool
```

expands to **`_Bool`**.

3 The remaining three macros are suitable for use in **`#if`** preprocessing directives. They are

```
true
```

which expands to the integer constant 1,

```
false
```

which expands to the integer constant 0, and

```
__bool_true_false_are_defined
```

which expands to the integer constant 1.

4 Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros **`bool`**, **`true`**, and **`false`**.[259)]

---

259) See ‘‘future library directions’’ (7.31.9).

7.19 Common definitions `<stddef.h>`

1 The header **`<stddef.h>`** defines the following macros and declares the following types. Some are also defined in other headers, as noted in their respective subclauses.

2 The types are

```
ptrdiff_t
```

which is the signed integer type of the result of subtracting two pointers;

```
size_t
```

which is the unsigned integer type of the result of the **`sizeof`** operator;

```
max_align_t
```

which is an object type whose alignment is as great as is supported by the implementation in all contexts; and

```
wchar_t
```

which is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero. Each member of the basic character set shall have a code value equal to its value when used as the lone character in an integer character constant if an implementation does not define **`__STDC_MB_MIGHT_NEQ_WC__`**.

3 The macros are

```
NULL
```

which expands to an implementation-defined null pointer constant; and

`offsetof(`*type*, *member-designator***`)`**

which expands to an integer constant expression that has type **`size_t`**, the value of which is the offset in bytes, to the structure member (designated by *member-designator*), from the beginning of its structure (designated by *type*). The type and member designator shall be such that given

`static` *type* **`t;`**

then the expression **`&(t.`***member-designator***`)`** evaluates to an address constant. (If the specified member is a bit-field, the behavior is undefined.)

Recommended practice

4 The types used for **`size_t`** and **`ptrdiff_t`** should not have an integer conversion rank greater than that of **`signed long int`** unless the implementation supports objects large enough to make this necessary. *

7.20 Integer types `<stdint.h>`

1 The header **`<stdint.h>`** declares sets of integer types having specified widths, and defines corresponding sets of macros.[260)] It also defines macros that specify limits of integer types corresponding to types defined in other standard headers.

2 Types are defined in the following categories:

— integer types having certain exact widths;

— integer types having at least certain specified widths;

— fastest integer types having at least certain specified widths;

— integer types wide enough to hold pointers to objects;

— integer types having greatest width.

(Some of these types may denote the same type.)

3 Corresponding macros specify limits of the declared types and construct suitable constants.

4 For each type described herein that the implementation provides,[261)] **`<stdint.h>`** shall declare that typedef name and define the associated macros. Conversely, for each type described herein that the implementation does not provide, **`<stdint.h>`** shall not declare that typedef name nor shall it define the associated macros. An implementation shall provide those types described as ‘‘required’’, but need not provide any of the others (described as ‘‘optional’’).

7.20.1 Integer types

1 When typedef names differing only in the absence or presence of the initial **`u`** are defined, they shall denote corresponding signed and unsigned types as described in 6.2.5; an implementation providing one of these corresponding types shall also provide the other.

2 In the following descriptions, the symbol *N* represents an unsigned decimal integer with no leading zeros (e.g., 8 or 24, but not 04 or 048).

---

260) See ‘‘future library directions’’ (7.31.10).

261) Some of these types may denote implementation-defined extended integer types.

7.20.1.1 Exact-width integer types

1 The typedef name **int***N***_t** designates a signed integer type with width *N*, no padding bits, and a two's complement representation. Thus, **int8_t** denotes such a signed integer type with a width of exactly 8 bits.

2 The typedef name **uint***N***_t** designates an unsigned integer type with width *N* and no padding bits. Thus, **uint24_t** denotes such an unsigned integer type with a width of exactly 24 bits.

3 These types are optional. However, if an implementation provides integer types with widths of 8, 16, 32, or 64 bits, no padding bits, and (for the signed types) that have a two's complement representation, it shall define the corresponding typedef names.

7.20.1.2 Minimum-width integer types

1 The typedef name **int_least***N***_t** designates a signed integer type with a width of at least *N*, such that no signed integer type with lesser size has at least the specified width. Thus, **int_least32_t** denotes a signed integer type with a width of at least 32 bits.

2 The typedef name **uint_least***N***_t** designates an unsigned integer type with a width of at least *N*, such that no unsigned integer type with lesser size has at least the specified width. Thus, **uint_least16_t** denotes an unsigned integer type with a width of at least 16 bits.

3 The following types are required:

```
int_least8_t          uint_least8_t
int_least16_t         uint_least16_t
int_least32_t         uint_least32_t
int_least64_t         uint_least64_t
```

All other types of this form are optional.

7.20.1.3 Fastest minimum-width integer types

1 Each of the following types designates an integer type that is usually fastest[262)] to operate with among all integer types that have at least the specified width.

2 The typedef name **int_fast***N***_t** designates the fastest signed integer type with a width of at least *N*. The typedef name **uint_fast***N***_t** designates the fastest unsigned integer type with a width of at least *N*.

262) The designated type is not guaranteed to be fastest for all purposes; if the implementation has no clear grounds for choosing one type over another, it will simply pick some integer type satisfying the signedness and width requirements.

3 The following types are required:

```
int_fast8_t           uint_fast8_t
int_fast16_t          uint_fast16_t
int_fast32_t          uint_fast32_t
int_fast64_t          uint_fast64_t
```

All other types of this form are optional.

7.20.1.4 Integer types capable of holding object pointers

1 The following type designates a signed integer type with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer:

```
intptr_t
```

The following type designates an unsigned integer type with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer:

```
uintptr_t
```

These types are optional.

7.20.1.5 Greatest-width integer types

1 The following type designates a signed integer type capable of representing any value of any signed integer type:

```
intmax_t
```

The following type designates an unsigned integer type capable of representing any value of any unsigned integer type:

```
uintmax_t
```

These types are required.

7.20.2 Limits of specified-width integer types

1 The following object-like macros specify the minimum and maximum limits of the types declared in **<stdint.h>**. Each macro name corresponds to a similar type name in 7.20.1.

2 Each instance of any defined macro shall be replaced by a constant expression suitable for use in **#if** preprocessing directives, and this expression shall have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Its implementation-defined value shall be equal to or greater in magnitude (absolute value) than the corresponding value given below, with the same sign, except where stated to be exactly the given value.

7.20.2.1 Limits of exact-width integer types

1 — minimum values of exact-width signed integer types

INTN**_MIN** exactly $-(2^{N-1})$

— maximum values of exact-width signed integer types

INTN**_MAX** exactly $2^{N-1} - 1$

— maximum values of exact-width unsigned integer types

UINTN**_MAX** exactly $2^{N} - 1$

7.20.2.2 Limits of minimum-width integer types

1 — minimum values of minimum-width signed integer types

INT_LEASTN**_MIN** $-(2^{N-1} - 1)$

— maximum values of minimum-width signed integer types

INT_LEASTN**_MAX** $2^{N-1} - 1$

— maximum values of minimum-width unsigned integer types

UINT_LEASTN**_MAX** $2^{N} - 1$

7.20.2.3 Limits of fastest minimum-width integer types

1 — minimum values of fastest minimum-width signed integer types

INT_FASTN**_MIN** $-(2^{N-1} - 1)$

— maximum values of fastest minimum-width signed integer types

INT_FASTN**_MAX** $2^{N-1} - 1$

— maximum values of fastest minimum-width unsigned integer types

UINT_FASTN**_MAX** $2^{N} - 1$

7.20.2.4 Limits of integer types capable of holding object pointers

1 — minimum value of pointer-holding signed integer type

INTPTR_MIN $-(2^{15} - 1)$

— maximum value of pointer-holding signed integer type

INTPTR_MAX $2^{15} - 1$

— maximum value of pointer-holding unsigned integer type

UINTPTR_MAX $2^{16} - 1$

7.20.2.5 Limits of greatest-width integer types

1 — minimum value of greatest-width signed integer type

`INTMAX_MIN`	$-(2^{63} - 1)$

— maximum value of greatest-width signed integer type

`INTMAX_MAX`	$2^{63} - 1$

— maximum value of greatest-width unsigned integer type

`UINTMAX_MAX`	$2^{64} - 1$

7.20.3 Limits of other integer types

1 The following object-like macros specify the minimum and maximum limits of integer types corresponding to types defined in other standard headers.

2 Each instance of these macros shall be replaced by a constant expression suitable for use in **`#if`** preprocessing directives, and this expression shall have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Its implementation-defined value shall be equal to or greater in magnitude (absolute value) than the corresponding value given below, with the same sign. An implementation shall define only the macros corresponding to those typedef names it actually provides.[263)]

— limits of **`ptrdiff_t`**

`PTRDIFF_MIN`	**`-65535`**
`PTRDIFF_MAX`	**`+65535`**

— limits of **`sig_atomic_t`**

`SIG_ATOMIC_MIN`	*see below*
`SIG_ATOMIC_MAX`	*see below*

— limit of **`size_t`**

`SIZE_MAX`	**`65535`**

— limits of **`wchar_t`**

`WCHAR_MIN`	*see below*
`WCHAR_MAX`	*see below*

— limits of **`wint_t`**

263) A freestanding implementation need not provide all of these types.

WINT_MIN	*see below*
WINT_MAX	*see below*

3 If **sig_atomic_t** (see 7.14) is defined as a signed integer type, the value of **SIG_ATOMIC_MIN** shall be no greater than −127 and the value of **SIG_ATOMIC_MAX** shall be no less than 127; otherwise, **sig_atomic_t** is defined as an unsigned integer type, and the value of **SIG_ATOMIC_MIN** shall be 0 and the value of **SIG_ATOMIC_MAX** shall be no less than 255.

4 If **wchar_t** (see 7.19) is defined as a signed integer type, the value of **WCHAR_MIN** shall be no greater than −127 and the value of **WCHAR_MAX** shall be no less than 127; otherwise, **wchar_t** is defined as an unsigned integer type, and the value of **WCHAR_MIN** shall be 0 and the value of **WCHAR_MAX** shall be no less than 255.[264)]

5 If **wint_t** (see 7.29) is defined as a signed integer type, the value of **WINT_MIN** shall be no greater than −32767 and the value of **WINT_MAX** shall be no less than 32767; otherwise, **wint_t** is defined as an unsigned integer type, and the value of **WINT_MIN** shall be 0 and the value of **WINT_MAX** shall be no less than 65535.

7.20.4 Macros for integer constants

1 The following function-like macros expand to integer constants suitable for initializing objects that have integer types corresponding to types defined in **<stdint.h>**. Each macro name corresponds to a similar type name in 7.20.1.2 or 7.20.1.5.

2 The argument in any instance of these macros shall be an unsuffixed integer constant (as defined in 6.4.4.1) with a value that does not exceed the limits for the corresponding type.

3 Each invocation of one of these macros shall expand to an integer constant expression suitable for use in **#if** preprocessing directives. The type of the expression shall have the same type as would an expression of the corresponding type converted according to the integer promotions. The value of the expression shall be that of the argument.

7.20.4.1 Macros for minimum-width integer constants

1 The macro **INT***N***_C(***value***)** shall expand to an integer constant expression corresponding to the type **int_least***N***_t**. The macro **UINT***N***_C(***value***)** shall expand to an integer constant expression corresponding to the type **uint_least***N***_t**. For example, if **uint_least64_t** is a name for the type **unsigned long long int**, then **UINT64_C(0x123)** might expand to the integer constant **0x123ULL**.

264) The values **WCHAR_MIN** and **WCHAR_MAX** do not necessarily correspond to members of the extended character set.

7.20.4.2 Macros for greatest-width integer constants

1 The following macro expands to an integer constant expression having the value specified by its argument and the type **`intmax_t`**:

`INTMAX_C(`*value***`)`**

The following macro expands to an integer constant expression having the value specified by its argument and the type **`uintmax_t`**:

`UINTMAX_C(`*value***`)`**

7.21 Input/output `<stdio.h>`

7.21.1 Introduction

1 The header **`<stdio.h>`** defines several macros, and declares three types and many functions for performing input and output.

2 The types declared are **`size_t`** (described in 7.19);

`FILE`

which is an object type capable of recording all the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an *error indicator* that records whether a read/write error has occurred, and an *end-of-file indicator* that records whether the end of the file has been reached; and

`fpos_t`

which is a complete object type other than an array type capable of recording all the information needed to specify uniquely every position within a file.

3 The macros are **`NULL`** (described in 7.19);

`_IOFBF`
`_IOLBF`
`_IONBF`

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the **`setvbuf`** function;

`BUFSIZ`

which expands to an integer constant expression that is the size of the buffer used by the **`setbuf`** function;

`EOF`

which expands to an integer constant expression, with type **`int`** and a negative value, that is returned by several functions to indicate *end-of-file*, that is, no more input from a stream;

`FOPEN_MAX`

which expands to an integer constant expression that is the minimum number of files that the implementation guarantees can be open simultaneously;

`FILENAME_MAX`

which expands to an integer constant expression that is the size needed for an array of **`char`** large enough to hold the longest file name string that the implementation guarantees can be opened;[265)]

L_tmpnam

which expands to an integer constant expression that is the size needed for an array of **char** large enough to hold a temporary file name string generated by the **tmpnam** function;

SEEK_CUR
SEEK_END
SEEK_SET

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the **fseek** function;

TMP_MAX

which expands to an integer constant expression that is the minimum number of unique file names that can be generated by the **tmpnam** function;

stderr
stdin
stdout

which are expressions of type ‘‘pointer to **FILE**’’ that point to the **FILE** objects associated, respectively, with the standard error, input, and output streams.

4 The header **<wchar.h>** declares a number of functions useful for wide character input and output. The wide character input/output functions described in that subclause provide operations analogous to most of those described here, except that the fundamental units internal to the program are wide characters. The external representation (in the file) is a sequence of ‘‘generalized’’ multibyte characters, as described further in 7.21.3.

5 The input/output functions are given the following collective terms:

— The *wide character input functions* — those functions described in 7.29 that perform input into wide characters and wide strings: **fgetwc**, **fgetws**, **getwc**, **getwchar**, **fwscanf**, **wscanf**, **vfwscanf**, and **vwscanf**.

— The *wide character output functions* — those functions described in 7.29 that perform output from wide characters and wide strings: **fputwc**, **fputws**, **putwc**, **putwchar**, **fwprintf**, **wprintf**, **vfwprintf**, and **vwprintf**.

---

265) If the implementation imposes no practical limit on the length of file name strings, the value of **FILENAME_MAX** should instead be the recommended size of an array intended to hold a file name string. Of course, file name string contents are subject to other system-specific constraints; therefore *all* possible strings of length **FILENAME_MAX** cannot be expected to be opened successfully.

— The *wide character input/output functions* — the union of the **ungetwc** function, the wide character input functions, and the wide character output functions.

— The *byte input/output functions* — those functions described in this subclause that perform input/output: **fgetc**, **fgets**, **fprintf**, **fputc**, **fputs**, **fread**, **fscanf**, **fwrite**, **getc**, **getchar**, **printf**, **putc**, **putchar**, **puts**, **scanf**, **ungetc**, **vfprintf**, **vfscanf**, **vprintf**, and **vscanf**.

Forward references: files (7.21.3), the **fseek** function (7.21.9.2), streams (7.21.2), the **tmpnam** function (7.21.4.4), **<wchar.h>** (7.29).

7.21.2 Streams

1 Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data *streams*, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported, for *text streams* and for *binary streams*.[266)]

2 A text stream is an ordered sequence of characters composed into *lines*, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is implementation-defined. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there need not be a one-to-one correspondence between the characters in a stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printing characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.

3 A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, have an implementation-defined number of null characters appended to the end of the stream.

4 Each stream has an *orientation*. After a stream is associated with an external file, but before any operations are performed on it, the stream is without orientation. Once a wide character input/output function has been applied to a stream without orientation, the

266) An implementation need not distinguish between text streams and binary streams. In such an implementation, there need be no new-line characters in a text stream nor any limit to the length of a line.

stream becomes a *wide-oriented stream*. Similarly, once a byte input/output function has been applied to a stream without orientation, the stream becomes a *byte-oriented stream*. Only a call to the **freopen** function or the **fwide** function can otherwise alter the orientation of a stream. (A successful call to **freopen** removes any orientation.)[267)]

5 Byte input/output functions shall not be applied to a wide-oriented stream and wide character input/output functions shall not be applied to a byte-oriented stream. The remaining stream operations do not affect, and are not affected by, a stream's orientation, except for the following additional restrictions:

— Binary wide-oriented streams have the file-positioning restrictions ascribed to both text and binary streams.

— For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide character output function can overwrite a partial multibyte character; any file contents beyond the byte(s) written are henceforth indeterminate.

6 Each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state of the stream. A successful call to **fgetpos** stores a representation of the value of this **mbstate_t** object as part of the value of the **fpos_t** object. A later successful call to **fsetpos** using the same stored **fpos_t** value restores the value of the associated **mbstate_t** object as well as the position within the controlled stream.

7 Each stream has an associated lock that is used to prevent data races when multiple threads of execution access a stream, and to restrict the interleaving of stream operations performed by multiple threads. Only one thread may hold this lock at a time. The lock is reentrant: a single thread may hold the lock multiple times at a given time.

8 All functions that read, write, position, or query the position of a stream lock the stream before accessing it. They release the lock associated with the stream when the access is complete.

Environmental limits

9 An implementation shall support text files with lines containing at least 254 characters, including the terminating new-line character. The value of the macro **BUFSIZ** shall be at least 256.

Forward references: the **freopen** function (7.21.5.4), the **fwide** function (7.29.3.5), **mbstate_t** (7.30.1), the **fgetpos** function (7.21.9.1), the **fsetpos** function (7.21.9.3).

267) The three predefined streams **stdin**, **stdout**, and **stderr** are unoriented at program startup.

7.21.3 Files

1 A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve *creating* a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a *file position indicator* associated with the stream is positioned at the start (character number zero) of the file, unless the file is opened with append mode in which case it is implementation-defined whether the file position indicator is initially positioned at the beginning or the end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file.

2 Binary files are not truncated, except as defined in 7.21.5.3. Whether a write on a text stream causes the associated file to be truncated beyond that point is implementation-defined.

3 When a stream is *unbuffered*, characters are intended to appear from the source or at the destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block. When a stream is *fully buffered*, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is *line buffered*, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered. Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line buffered stream that requires the transmission of characters from the host environment. Support for these characteristics is implementation-defined, and may be affected via the **`setbuf`** and **`setvbuf`** functions.

4 A file may be disassociated from a controlling stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The value of a pointer to a **`FILE`** object is indeterminate after the associated file is closed (including the standard text streams). Whether a file of zero length (on which no characters have been written by an output stream) actually exists is implementation-defined.

5 The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the **`main`** function returns to its original caller, or if the **`exit`** function is called, all open files are closed (hence all output streams are flushed) before program termination. Other paths to program termination, such as calling the **`abort`** function, need not close all files properly.

6 The address of the **`FILE`** object used to control a stream may be significant; a copy of a **`FILE`** object need not serve in place of the original.

7 At program startup, three text streams are predefined and need not be opened explicitly — *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). As initially opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

8 Functions that open additional (nontemporary) files require a *file name*, which is a string. The rules for composing valid file names are implementation-defined. Whether the same file can be simultaneously open multiple times is also implementation-defined.

9 Although both text and binary wide-oriented streams are conceptually sequences of wide characters, the external file associated with a wide-oriented stream is a sequence of multibyte characters, generalized as follows:

— Multibyte encodings within files may contain embedded null bytes (unlike multibyte encodings valid for use internal to the program).

— A file need not begin nor end in the initial shift state.[268)]

10 Moreover, the encodings used for multibyte characters may differ among files. Both the nature and choice of such encodings are implementation-defined.

11 The wide character input functions read multibyte characters from the stream and convert them to wide characters as if they were read by successive calls to the **fgetwc** function. Each conversion occurs as if by a call to the **mbrtowc** function, with the conversion state described by the stream's own **mbstate_t** object. The byte input functions read characters from the stream as if by successive calls to the **fgetc** function.

12 The wide character output functions convert wide characters to multibyte characters and write them to the stream as if they were written by successive calls to the **fputwc** function. Each conversion occurs as if by a call to the **wcrtomb** function, with the conversion state described by the stream's own **mbstate_t** object. The byte output functions write characters to the stream as if by successive calls to the **fputc** function.

13 In some cases, some of the byte input/output functions also perform conversions between multibyte characters and wide characters. These conversions also occur as if by calls to the **mbrtowc** and **wcrtomb** functions.

14 An *encoding error* occurs if the character sequence presented to the underlying **mbrtowc** function does not form a valid (generalized) multibyte character, or if the code value passed to the underlying **wcrtomb** does not correspond to a valid (generalized)

268) Setting the file position indicator to end-of-file, as with **fseek(file, 0, SEEK_END)**, has undefined behavior for a binary stream (because of possible trailing null characters) or for any stream with state-dependent encoding that does not assuredly end in the initial shift state.

multibyte character. The wide character input/output functions and the byte input/output functions store the value of the macro **EILSEQ** in **errno** if and only if an encoding error occurs.

Environmental limits

15 The value of **FOPEN_MAX** shall be at least eight, including the three standard text streams.

Forward references: the **exit** function (7.22.4.4), the **fgetc** function (7.21.7.1), the **fopen** function (7.21.5.3), the **fputc** function (7.21.7.3), the **setbuf** function (7.21.5.5), the **setvbuf** function (7.21.5.6), the **fgetwc** function (7.29.3.1), the **fputwc** function (7.29.3.3), conversion state (7.29.6), the **mbrtowc** function (7.29.6.3.2), the **wcrtomb** function (7.29.6.3.3).

7.21.4 Operations on files

7.21.4.1 The remove function

Synopsis

1
```
#include <stdio.h>
int remove(const char *filename);
```

Description

2 The **remove** function causes the file whose name is the string pointed to by **filename** to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the behavior of the **remove** function is implementation-defined.

Returns

3 The **remove** function returns zero if the operation succeeds, nonzero if it fails.

7.21.4.2 The rename function

Synopsis

1
```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Description

2 The **rename** function causes the file whose name is the string pointed to by **old** to be henceforth known by the name given by the string pointed to by **new**. The file named **old** is no longer accessible by that name. If a file named by the string pointed to by **new** exists prior to the call to the **rename** function, the behavior is implementation-defined.

Returns

3 The **`rename`** function returns zero if the operation succeeds, nonzero if it fails,[269] in which case if the file existed previously it is still known by its original name.

7.21.4.3 The `tmpfile` function

Synopsis

1
```
#include <stdio.h>
FILE *tmpfile(void);
```

Description

2 The **`tmpfile`** function creates a temporary binary file that is different from any other existing file and that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with **`"wb+"`** mode.

Recommended practice

3 It should be possible to open at least **`TMP_MAX`** temporary files during the lifetime of the program (this limit may be shared with **`tmpnam`**) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files (**`FOPEN_MAX`**).

Returns

4 The **`tmpfile`** function returns a pointer to the stream of the file that it created. If the file cannot be created, the **`tmpfile`** function returns a null pointer.

Forward references: the **`fopen`** function (7.21.5.3).

7.21.4.4 The `tmpnam` function

Synopsis

1
```
#include <stdio.h>
char *tmpnam(char *s);
```

Description

2 The **`tmpnam`** function generates a string that is a valid file name and that is not the same as the name of an existing file.[270] The function is potentially capable of generating at

269) Among the reasons the implementation may cause the **`rename`** function to fail are that the file is open or that it is necessary to copy its contents to effectuate its renaming.

270) Files created using strings generated by the **`tmpnam`** function are temporary only in the sense that their names should not collide with those generated by conventional naming rules for the implementation. It is still necessary to use the **`remove`** function to remove such files when their use is ended, and before program termination.

least **TMP_MAX** different strings, but any or all of them may already be in use by existing files and thus not be suitable return values.

3 The **tmpnam** function generates a different string each time it is called.

4 Calls to the **tmpnam** function with a null pointer argument may introduce data races with each other. The implementation shall behave as if no library function calls the **tmpnam** function.

Returns

5 If no suitable string can be generated, the **tmpnam** function returns a null pointer. Otherwise, if the argument is a null pointer, the **tmpnam** function leaves its result in an internal static object and returns a pointer to that object (subsequent calls to the **tmpnam** function may modify the same object). If the argument is not a null pointer, it is assumed to point to an array of at least **L_tmpnam** **char**s; the **tmpnam** function writes its result in that array and returns the argument as its value.

Environmental limits

6 The value of the macro **TMP_MAX** shall be at least 25.

7.21.5 File access functions

7.21.5.1 The fclose function

Synopsis

1
```
#include <stdio.h>
int fclose(FILE *stream);
```

Description

2 A successful call to the **fclose** function causes the stream pointed to by **stream** to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. Whether or not the call succeeds, the stream is disassociated from the file and any buffer set by the **setbuf** or **setvbuf** function is disassociated from the stream (and deallocated if it was automatically allocated).

Returns

3 The **fclose** function returns zero if the stream was successfully closed, or **EOF** if any errors were detected.

7.21.5.2 The `fflush` function

Synopsis

1
```
#include <stdio.h>
int fflush(FILE *stream);
```

Description

2 If **`stream`** points to an output stream or an update stream in which the most recent operation was not input, the **`fflush`** function causes any unwritten data for that stream to be delivered to the host environment to be written to the file; otherwise, the behavior is undefined.

3 If **`stream`** is a null pointer, the **`fflush`** function performs this flushing action on all streams for which the behavior is defined above.

Returns

4 The **`fflush`** function sets the error indicator for the stream and returns **`EOF`** if a write error occurs, otherwise it returns zero.

Forward references: the **`fopen`** function (7.21.5.3).

7.21.5.3 The `fopen` function

Synopsis

1
```
#include <stdio.h>
FILE *fopen(const char * restrict filename,
     const char * restrict mode);
```

Description

2 The **`fopen`** function opens the file whose name is the string pointed to by **`filename`**, and associates a stream with it.

3 The argument **`mode`** points to a string. If the string is one of the following, the file is open in the indicated mode. Otherwise, the behavior is undefined.[271)]

`r`	open text file for reading
`w`	truncate to zero length or create text file for writing
`wx`	create text file for writing
`a`	append; open or create text file for writing at end-of-file
`rb`	open binary file for reading
`wb`	truncate to zero length or create binary file for writing

271) If the string begins with one of the above sequences, the implementation might choose to ignore the remaining characters, or it might use them to select different kinds of a file (some of which might not conform to the properties in 7.21.2).

wbx	create binary file for writing
ab	append; open or create binary file for writing at end-of-file
r+	open text file for update (reading and writing)
w+	truncate to zero length or create text file for update
w+x	create text file for update
a+	append; open or create text file for update, writing at end-of-file
r+b *or* **rb+**	open binary file for update (reading and writing)
w+b *or* **wb+**	truncate to zero length or create binary file for update
w+bx *or* **wb+x**	create binary file for update
a+b *or* **ab+**	append; open or create binary file for update, writing at end-of-file

4 Opening a file with read mode (**'r'** as the first character in the **mode** argument) fails if the file does not exist or cannot be read.

5 Opening a file with exclusive mode (**'x'** as the last character in the **mode** argument) fails if the file already exists or cannot be created. Otherwise, the file is created with exclusive (also known as non-shared) access to the extent that the underlying system supports exclusive access.

6 Opening a file with append mode (**'a'** as the first character in the **mode** argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to the **fseek** function. In some implementations, opening a binary file with append mode (**'b'** as the second or third character in the above list of **mode** argument values) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.

7 When a file is opened with update mode (**'+'** as the second or third character in the above list of **mode** argument values), both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the **fflush** function or to a file positioning function (**fseek**, **fsetpos**, or **rewind**), and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.

8 When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

Returns

9 The **fopen** function returns a pointer to the object controlling the stream. If the open operation fails, **fopen** returns a null pointer.

Forward references: file positioning functions (7.21.9).

7.21.5.4 The `freopen` function

Synopsis

1
```
#include <stdio.h>
FILE *freopen(const char * restrict filename,
     const char * restrict mode,
     FILE * restrict stream);
```

Description

2 The **freopen** function opens the file whose name is the string pointed to by **filename** and associates the stream pointed to by **stream** with it. The **mode** argument is used just as in the **fopen** function.[272)]

3 If **filename** is a null pointer, the **freopen** function attempts to change the mode of the stream to that specified by **mode**, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

4 The **freopen** function first attempts to close any file that is associated with the specified stream. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared.

Returns

5 The **freopen** function returns a null pointer if the open operation fails. Otherwise, **freopen** returns the value of **stream**.

7.21.5.5 The `setbuf` function

Synopsis

1
```
#include <stdio.h>
void setbuf(FILE * restrict stream,
     char * restrict buf);
```

Description

2 Except that it returns no value, the **setbuf** function is equivalent to the **setvbuf** function invoked with the values **_IOFBF** for **mode** and **BUFSIZ** for **size**, or (if **buf** is a null pointer), with the value **_IONBF** for **mode**.

272) The primary use of the **freopen** function is to change the file associated with a standard text stream (**stderr**, **stdin**, or **stdout**), as those identifiers need not be modifiable lvalues to which the value returned by the **fopen** function may be assigned.

Returns

3 The **setbuf** function returns no value.

Forward references: the **setvbuf** function (7.21.5.6).

7.21.5.6 The setvbuf function

Synopsis

1
```
#include <stdio.h>
int setvbuf(FILE * restrict stream,
     char * restrict buf,
     int mode, size_t size);
```

Description

2 The **setvbuf** function may be used only after the stream pointed to by **stream** has been associated with an open file and before any other operation (other than an unsuccessful call to **setvbuf**) is performed on the stream. The argument **mode** determines how **stream** will be buffered, as follows: **_IOFBF** causes input/output to be fully buffered; **_IOLBF** causes input/output to be line buffered; **_IONBF** causes input/output to be unbuffered. If **buf** is not a null pointer, the array it points to may be used instead of a buffer allocated by the **setvbuf** function[273] and the argument **size** specifies the size of the array; otherwise, **size** may determine the size of a buffer allocated by the **setvbuf** function. The contents of the array at any time are indeterminate.

Returns

3 The **setvbuf** function returns zero on success, or nonzero if an invalid value is given for **mode** or if the request cannot be honored.

273) The buffer has to have a lifetime at least as great as the open stream, so the stream should be closed before a buffer that has automatic storage duration is deallocated upon block exit.

7.21.6 Formatted input/output functions

1 The formatted input/output functions shall behave as if there is a sequence point after the actions associated with each specifier.[274)]

7.21.6.1 The `fprintf` function

Synopsis

1
```
#include <stdio.h>
int fprintf(FILE * restrict stream,
     const char * restrict format, ...);
```

Description

2 The **`fprintf`** function writes output to the stream pointed to by **`stream`**, under control of the string pointed to by **`format`** that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **`fprintf`** function returns when the end of the format string is encountered.

3 The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not **`%`**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

4 Each conversion specification is introduced by the character **`%`**. After the **`%`**, the following appear in sequence:

— Zero or more *flags* (in any order) that modify the meaning of the conversion specification.

— An optional minimum *field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk **`*`** (described later) or a nonnegative decimal integer.[275)]

— An optional *precision* that gives the minimum number of digits to appear for the **`d`**, **`i`**, **`o`**, **`u`**, **`x`**, and **`X`** conversions, the number of digits to appear after the decimal-point character for **`a`**, **`A`**, **`e`**, **`E`**, **`f`**, and **`F`** conversions, the maximum number of significant digits for the **`g`** and **`G`** conversions, or the maximum number of bytes to be written for

---

274) The **`fprintf`** functions perform writes to memory for the **`%n`** specifier.

275) Note that **`0`** is taken as a flag, not as the beginning of a field width.

s conversions. The precision takes the form of a period (**.**) followed either by an asterisk ***** (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

— An optional *length modifier* that specifies the size of the argument.

— A *conversion specifier* character that specifies the type of conversion to be applied.

5 As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an **int** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a **-** flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

6 The flag characters and their meanings are:

- The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

+ The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified.)[276)]

space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the *space* and **+** flags both appear, the *space* flag is ignored.

The result is converted to an ''alternative form''. For **o** conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For **x** (or **X**) conversion, a nonzero result has **0x** (or **0X**) prefixed to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For **g** and **G** conversions, trailing zeros are *not* removed from the result. For other conversions, the behavior is undefined.

0 For **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the **0** and **-** flags both appear, the **0** flag is ignored. For **d**, **i**, **o**, **u**, **x**, and **X**

276) The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

conversions, if a precision is specified, the **0** flag is ignored. For other conversions, the behavior is undefined.

7 The length modifiers and their meanings are:

hh Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **signed char** or **unsigned char** before printing); or that a following **n** conversion specifier applies to a pointer to a **signed char** argument.

h Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **short int** or **unsigned short int** before printing); or that a following **n** conversion specifier applies to a pointer to a **short int** argument.

l (ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long int** or **unsigned long int** argument; that a following **n** conversion specifier applies to a pointer to a **long int** argument; that a following **c** conversion specifier applies to a **wint_t** argument; that a following **s** conversion specifier applies to a pointer to a **wchar_t** argument; or has no effect on a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier.

ll (ell-ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long long int** or **unsigned long long int** argument; or that a following **n** conversion specifier applies to a pointer to a **long long int** argument.

j Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to an **intmax_t** or **uintmax_t** argument; or that a following **n** conversion specifier applies to a pointer to an **intmax_t** argument.

z Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **size_t** or the corresponding signed integer type argument; or that a following **n** conversion specifier applies to a pointer to a signed integer type corresponding to **size_t** argument.

t Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **ptrdiff_t** or the corresponding unsigned integer type argument; or that a following **n** conversion specifier applies to a pointer to a **ptrdiff_t** argument.

`L` Specifies that a following **`a`**, **`A`**, **`e`**, **`E`**, **`f`**, **`F`**, **`g`**, or **`G`** conversion specifier applies to a **`long double`** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

8 The conversion specifiers and their meanings are:

`d,i` The **`int`** argument is converted to signed decimal in the style *[*`−`*]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`o,u,x,X` The **`unsigned int`** argument is converted to unsigned octal (**`o`**), unsigned decimal (**`u`**), or unsigned hexadecimal notation (**`x`** or **`X`**) in the style *dddd*; the letters **`abcdef`** are used for **`x`** conversion and the letters **`ABCDEF`** for **`X`** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`f,F` A **`double`** argument representing a floating-point number is converted to decimal notation in the style *[*`−`*]ddd*`.`*ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **`#`** flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

A **`double`** argument representing an infinity is converted in one of the styles *[*`-`*]***`inf`** or *[*`-`*]***`infinity`** — which style is implementation-defined. A **`double`** argument representing a NaN is converted in one of the styles *[*`-`*]***`nan`** or *[*`-`*]***`nan(`***n-char-sequence***`)`** — which style, and the meaning of any *n-char-sequence*, is implementation-defined. The **`F`** conversion specifier produces **`INF`**, **`INFINITY`**, or **`NAN`** instead of **`inf`**, **`infinity`**, or **`nan`**, respectively.[277)]

`e,E` A **`double`** argument representing a floating-point number is converted in the style *[*`−`*]d*`.`*ddd* **`e`**±*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as

277) When applied to infinite and NaN values, the **`-`**, **`+`**, and *space* flag characters have their usual meaning; the **`#`** and **`0`** flag characters have no effect.

6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

g,G A **double** argument representing a floating-point number is converted in style **f** or **e** (or in style **F** or **E** in the case of a **G** conversion specifier), depending on the value converted and the precision. Let P equal the precision if nonzero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style **E** would have an exponent of X:

— if $P > X \geq -4$, the conversion is with style **f** (or **F**) and precision $P - (X + 1)$.

— otherwise, the conversion is with style **e** (or **E**) and precision $P - 1$.

Finally, unless the **#** flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point character is removed if there is no fractional portion remaining.

A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

a,A A **double** argument representing a floating-point number is converted in the style *[*-*]***0x***h***.***hhhh***p**±*d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character[278] and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and **FLT_RADIX** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **FLT_RADIX** is not a power of 2, then the precision is sufficient to

278) Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble (4-bit) boundaries.

distinguish[279)] values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. The letters **abcdef** are used for **a** conversion and the letters **ABCDEF** for **A** conversion. The **A** conversion specifier produces a number with **X** and **P** instead of **x** and **p**. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

c If no **l** length modifier is present, the **int** argument is converted to an **unsigned char**, and the resulting character is written.

If an **l** length modifier is present, the **wint_t** argument is converted as if by an **ls** conversion specification with no precision and an argument that points to the initial element of a two-element array of **wchar_t**, the first element containing the **wint_t** argument to the **lc** conversion specification and the second a null wide character.

s If no **l** length modifier is present, the argument shall be a pointer to the initial element of an array of character type.[280)] Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many bytes are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

If an **l** length modifier is present, the argument shall be a pointer to the initial element of an array of **wchar_t** type. Wide characters from the array are converted to multibyte characters (each as if by a call to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many bytes are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multibyte character sequence length given by

279) The precision p is sufficient to distinguish values of the source type if $16^{p-1} > b^n$ where b is **FLT_RADIX** and n is the number of base-b digits in the significand of the source type. A smaller p might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point character.

280) No special provisions are made for multibyte characters.

the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written.[281)]

p The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printing characters, in an implementation-defined manner.

n The argument shall be a pointer to signed integer into which is *written* the number of characters written to the output stream so far by this call to **fprintf**. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

% A **%** character is written. No argument is converted. The complete conversion specification shall be **%%**.

9 If a conversion specification is invalid, the behavior is undefined.[282)] If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

10 In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

11 For **a** and **A** conversions, if **FLT_RADIX** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

12 For **a** and **A** conversions, if **FLT_RADIX** is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

13 For **e**, **E**, **f**, **F**, **g**, and **G** conversions, if the number of significant decimal digits is at most **DECIMAL_DIG**, then the result should be correctly rounded.[283)] If the number of significant decimal digits is more than **DECIMAL_DIG** but the source value is exactly representable with **DECIMAL_DIG** digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having **DECIMAL_DIG** significant digits; the value

281) Redundant shift sequences may result if multibyte characters have a state-dependent encoding.

282) See ‘‘future library directions’’ (7.31.11).

283) For binary-to-decimal conversion, the result format’s values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

of the resultant decimal string D should satisfy $L \leq D \leq U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

Returns

14 The **`fprintf`** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Environmental limits

15 The number of characters that can be produced by any single conversion shall be at least 4095.

16 EXAMPLE 1 To print a date and time in the form ‘‘Sunday, July 3, 10:02’’ followed by π to five decimal places:

```
#include <math.h>
#include <stdio.h>
/* ... */
char *weekday, *month;     // pointers to strings
int day, hour, min;
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
      weekday, month, day, hour, min);
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

17 EXAMPLE 2 In this example, multibyte characters do not have a state-dependent encoding, and the members of the extended character set that consist of more than one byte each consist of exactly two bytes, the first of which is denoted here by a □ and the second by an uppercase letter.

18 Given the following wide string with length seven,

```
static wchar_t wstr[] = L"□X□Yabc□Z□W";
```

the seven calls

```
fprintf(stdout, "|1234567890123|\n");
fprintf(stdout, "|%13ls|\n", wstr);
fprintf(stdout, "|%-13.9ls|\n", wstr);
fprintf(stdout, "|%13.10ls|\n", wstr);
fprintf(stdout, "|%13.11ls|\n", wstr);
fprintf(stdout, "|%13.15ls|\n", &wstr[2]);
fprintf(stdout, "|%13lc|\n", (wint_t) wstr[5]);
```

will print the following seven lines:

```
|1234567890123|
|  □X□Yabc□Z□W|
|□X□Yabc□Z    |
|    □X□Yabc□Z|
|  □X□Yabc□Z□W|
|      abc□Z□W|
|           □Z|
```

Forward references: conversion state (7.29.6), the **`wcrtomb`** function (7.29.6.3.3).

7.21.6.2 The `fscanf` function

Synopsis

1
```
#include <stdio.h>
int fscanf(FILE * restrict stream,
     const char * restrict format, ...);
```

Description

2 The **`fscanf`** function reads input from the stream pointed to by **`stream`**, under control of the string pointed to by **`format`** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

3 The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters, an ordinary multibyte character (neither **`%`** nor a white-space character), or a conversion specification. Each conversion specification is introduced by the character **`%`**. After the **`%`**, the following appear in sequence:

— An optional assignment-suppressing character **`*`**.

— An optional decimal integer greater than zero that specifies the maximum field width (in characters).

— An optional *length modifier* that specifies the size of the receiving object.

— A *conversion specifier* character that specifies the type of conversion to be applied.

4 The **`fscanf`** function executes each directive of the format in turn. When all directives have been executed, or if a directive fails (as detailed below), the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

5 A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read. The directive never fails.

6 A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive, the directive fails and the differing and subsequent characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive fails.

7 A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the

following steps:

8 Input white-space characters (as specified by the **isspace** function) are skipped, unless the specification includes a **[**, **c**, or **n** specifier.[284)]

9 An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence.[285)] The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

10 Except in the case of a **%** specifier, the input item (or, in the case of a **%n** directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *****, the result of the conversion is placed in the object pointed to by the first argument following the **format** argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

11 The length modifiers and their meanings are:

hh Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.

h Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **short int** or **unsigned short int**.

l (ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **long int** or **unsigned long int**; that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to an argument with type pointer to **double**; or that a following **c**, **s**, or **[** conversion specifier applies to an argument with type pointer to **wchar_t**.

ll (ell-ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **long long int** or **unsigned long long int**.

---

284) These white-space characters are not counted against a specified field width.

285) **fscanf** pushes back at most one input character onto the input stream. Therefore, some sequences that are acceptable to **strtod**, **strtol**, etc., are unacceptable to **fscanf**.

j Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **intmax_t** or **uintmax_t**.

z Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **size_t** or the corresponding signed integer type.

t Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **ptrdiff_t** or the corresponding unsigned integer type.

L Specifies that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to an argument with type pointer to **long double**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

12 The conversion specifiers and their meanings are:

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to signed integer.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 0 for the **base** argument. The corresponding argument shall be a pointer to signed integer.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 8 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 16 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

a,e,f,g Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the **strtod** function. The corresponding argument shall be a pointer to floating.

c Matches a sequence of characters of exactly the number specified by the field width (1 if no field width is present in the directive).[286)]

If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the resulting sequence of wide characters. No null wide character is added.

s Matches a sequence of non-white-space characters.[286)]

If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

[Matches a nonempty sequence of characters from a set of expected characters (the *scanset*).[286)]

If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero

286) No special provisions are made for multibyte characters in the matching rules used by the **c**, **s**, and **[** conversion specifiers — the extent of the input field is determined on a byte-by-byte basis. The resulting field is nevertheless a sequence of multibyte characters that begins in the initial shift state.

before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent characters in the **format** string, up to and including the matching right bracket (**]**). The characters between the brackets (the *scanlist*) compose the scanset, unless the character after the left bracket is a circumflex (**^**), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with **[]** or **[^]**, the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a **-** character is in the scanlist and is not the first, nor the second where the first character is a **^**, nor the last character, the behavior is implementation-defined.

p Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the **%p** conversion of the **fprintf** function. The corresponding argument shall be a pointer to a pointer to **void**. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the **%p** conversion is undefined.

n No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the **fscanf** function. Execution of a **%n** directive does not increment the assignment count returned at the completion of execution of the **fscanf** function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

% Matches a single **%** character; no conversion or assignment occurs. The complete conversion specification shall be **%%**.

13 If a conversion specification is invalid, the behavior is undefined.[287)]

14 The conversion specifiers **A**, **E**, **F**, **G**, and **X** are also valid and behave the same as, respectively, **a**, **e**, **f**, **g**, and **x**.

287) See ‘‘future library directions’’ (7.31.11).

15 Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** directive.

Returns

16 The **fscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

17 EXAMPLE 1 The call:

```
#include <stdio.h>
/* ... */
int n, i; float x; char name[50];
n = fscanf(stdin, "%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to **n** the value 3, to **i** the value 25, to **x** the value 5.432, and to **name** the sequence **thompson\0**.

18 EXAMPLE 2 The call:

```
#include <stdio.h>
/* ... */
int i; float x; char name[50];
fscanf(stdin, "%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign to **i** the value 56 and to **x** the value 789.0, will skip **0123**, and will assign to **name** the sequence **56\0**. The next character read from the input stream will be **a**.

19 EXAMPLE 3 To accept repeatedly from **stdin** a quantity, a unit of measure, and an item name:

```
#include <stdio.h>
/* ... */
int count; float quant; char units[21], item[21];
do {
      count = fscanf(stdin, "%f%20s of %20s", &quant, units, item);
      fscanf(stdin,"%*[^\n]");
} while (!feof(stdin) && !ferror(stdin));
```

20 If the **stdin** stream contains the following lines:

```
2 quarts of oil
-12.8degrees Celsius
lots of luck
10.0LBS     of
dirt
100ergs of energy
```

the execution of the above example will be analogous to the following assignments:

```
quant = 2; strcpy(units, "quarts"); strcpy(item, "oil");
count = 3;
quant = -12.8; strcpy(units, "degrees");
count = 2; // "C" fails to match "o"
count = 0; // "l" fails to match "%f"
quant = 10.0; strcpy(units, "LBS"); strcpy(item, "dirt");
count = 3;
count = 0; // "100e" fails to match "%f"
count = EOF;
```

21 EXAMPLE 4 In:

```
#include <stdio.h>
/* ... */
int d1, d2, n1, n2, i;
i = sscanf("123", "%d%n%n%d", &d1, &n1, &n2, &d2);
```

the value 123 is assigned to **d1** and the value 3 to **n1**. Because **%n** can never get an input failure, the value of 3 is also assigned to **n2**. The value of **d2** is not affected. The value 1 is assigned to **i**.

22 EXAMPLE 5 The call:

```
#include <stdio.h>
/* ... */
int n, i;
n = sscanf("foo  %  bar  42", "foo%%bar%d", &i);
```

will assign to **n** the value 1 and to **i** the value 42 because input white-space characters are skipped for both the **%** and **d** conversion specifiers.

23 EXAMPLE 6 In these examples, multibyte characters do have a state-dependent encoding, and the members of the extended character set that consist of more than one byte each consist of exactly two bytes, the first of which is denoted here by a □ and the second by an uppercase letter, but are only recognized as such when in the alternate shift state. The shift sequences are denoted by ↑ and ↓, in which the first causes entry into the alternate shift state.

24 After the call:

```
#include <stdio.h>
/* ... */
char str[50];
fscanf(stdin, "a%s", str);
```

with the input line:

```
a↑□X□Y↓ bc
```

str will contain ↑**□X□Y**↓**\0** assuming that none of the bytes of the shift sequences (or of the multibyte characters, in the more general case) appears to be a single-byte white-space character.

25 In contrast, after the call:

```
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a%ls", wstr);
```

with the same input line, **wstr** will contain the two wide characters that correspond to □**X** and □**Y** and a terminating null wide character.

26 However, the call:

```
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a↑□X↓%ls", wstr);
```

with the same input line will return zero due to a matching failure against the ↓ sequence in the format string.

27 Assuming that the first byte of the multibyte character □**X** is the same as the first byte of the multibyte character □**Y**, after the call:

```
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a↑□Y↓%ls", wstr);
```

with the same input line, zero will again be returned, but **stdin** will be left with a partially consumed multibyte character.

Forward references: the **strtod**, **strtof**, and **strtold** functions (7.22.1.3), the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.22.1.4), conversion state (7.29.6), the **wcrtomb** function (7.29.6.3.3).

7.21.6.3 The printf function

Synopsis

1

```
#include <stdio.h>
int printf(const char * restrict format, ...);
```

Description

2 The **printf** function is equivalent to **fprintf** with the argument **stdout** interposed before the arguments to **printf**.

Returns

3 The **printf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

7.21.6.4 The `scanf` function

Synopsis

1
```
#include <stdio.h>
int scanf(const char * restrict format, ...);
```

Description

2 The **`scanf`** function is equivalent to **`fscanf`** with the argument **`stdin`** interposed before the arguments to **`scanf`**.

Returns

3 The **`scanf`** function returns the value of the macro **`EOF`** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **`scanf`** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.6.5 The `snprintf` function

Synopsis

1
```
#include <stdio.h>
int snprintf(char * restrict s, size_t n,
     const char * restrict format, ...);
```

Description

2 The **`snprintf`** function is equivalent to **`fprintf`**, except that the output is written into an array (specified by argument **`s`**) rather than to a stream. If **`n`** is zero, nothing is written, and **`s`** may be a null pointer. Otherwise, output characters beyond the **`n-1`**st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **`snprintf`** function returns the number of characters that would have been written had **`n`** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **`n`**.

7.21.6.6 The `sprintf` function

Synopsis

1
```
#include <stdio.h>
int sprintf(char * restrict s,
     const char * restrict format, ...);
```

Description

2 The **sprintf** function is equivalent to **fprintf**, except that the output is written into an array (specified by the argument **s**) rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **sprintf** function returns the number of characters written in the array, not counting the terminating null character, or a negative value if an encoding error occurred.

7.21.6.7 The sscanf function

Synopsis

1
```
#include <stdio.h>
int sscanf(const char * restrict s,
     const char * restrict format, ...);
```

Description

2 The **sscanf** function is equivalent to **fscanf**, except that input is obtained from a string (specified by the argument **s**) rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the **fscanf** function. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **sscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **sscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.6.8 The vfprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE * restrict stream,
     const char * restrict format,
     va_list arg);
```

Description

2 The **vfprintf** function is equivalent to **fprintf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfprintf** function does not invoke the

va_end macro.[288)]

Returns

3 The **vfprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

4 EXAMPLE The following shows the use of the **vfprintf** function in a general error-reporting routine.

```
#include <stdarg.h>
#include <stdio.h>

void error(char *function_name, char *format, ...)
{
      va_list args;

      va_start(args, format);
      // print out name of function causing error
      fprintf(stderr, "ERROR in %s: ", function_name);
      // print out remainder of message
      vfprintf(stderr, format, args);
      va_end(args);
}
```

7.21.6.9 The vfscanf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vfscanf(FILE * restrict stream,
     const char * restrict format,
     va_list arg);
```

Description

2 The **vfscanf** function is equivalent to **fscanf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfscanf** function does not invoke the **va_end** macro.[288)]

Returns

3 The **vfscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vfscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

288) As the functions **vfprintf**, **vfscanf**, **vprintf**, **vscanf**, **vsnprintf**, **vsprintf**, and **vsscanf** invoke the **va_arg** macro, the value of **arg** after the return is indeterminate.

7.21.6.10 The `vprintf` function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char * restrict format,
     va_list arg);
```

Description

2 The **vprintf** function is equivalent to **printf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vprintf** function does not invoke the **va_end** macro.[288)]

Returns

3 The **vprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

7.21.6.11 The `vscanf` function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vscanf(const char * restrict format,
     va_list arg);
```

Description

2 The **vscanf** function is equivalent to **scanf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vscanf** function does not invoke the **va_end** macro.[288)]

Returns

3 The **vscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.6.12 The `vsnprintf` function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsnprintf(char * restrict s, size_t n,
     const char * restrict format,
     va_list arg);
```

Description

2 The **`vsnprintf`** function is equivalent to **`snprintf`**, with the variable argument list replaced by **`arg`**, which shall have been initialized by the **`va_start`** macro (and possibly subsequent **`va_arg`** calls). The **`vsnprintf`** function does not invoke the **`va_end`** macro.[288)] If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **`vsnprintf`** function returns the number of characters that would have been written had **`n`** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **`n`**.

7.21.6.13 The `vsprintf` function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char * restrict s,
     const char * restrict format,
     va_list arg);
```

Description

2 The **`vsprintf`** function is equivalent to **`sprintf`**, with the variable argument list replaced by **`arg`**, which shall have been initialized by the **`va_start`** macro (and possibly subsequent **`va_arg`** calls). The **`vsprintf`** function does not invoke the **`va_end`** macro.[288)] If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **`vsprintf`** function returns the number of characters written in the array, not counting the terminating null character, or a negative value if an encoding error occurred.

7.21.6.14 The vsscanf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsscanf(const char * restrict s,
     const char * restrict format,
     va_list arg);
```

Description

2 The **vsscanf** function is equivalent to **sscanf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsscanf** function does not invoke the **va_end** macro.[288)]

Returns

3 The **vsscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vsscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.7 Character input/output functions

7.21.7.1 The fgetc function

Synopsis

1
```
#include <stdio.h>
int fgetc(FILE *stream);
```

Description

2 If the end-of-file indicator for the input stream pointed to by **stream** is not set and a next character is present, the **fgetc** function obtains that character as an **unsigned char** converted to an **int** and advances the associated file position indicator for the stream (if defined).

Returns

3 If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream is set and the **fgetc** function returns **EOF**. Otherwise, the **fgetc** function returns the next character from the input stream pointed to by **stream**. If a read error occurs, the error indicator for the stream is set and the **fgetc** function returns **EOF**.[289)]

289) An end-of-file and a read error can be distinguished by use of the **feof** and **ferror** functions.

7.21.7.2 The `fgets` function

Synopsis

1

```
#include <stdio.h>
char *fgets(char * restrict s, int n,
     FILE * restrict stream);
```

Description

2 The **`fgets`** function reads at most one less than the number of characters specified by **`n`** from the stream pointed to by **`stream`** into the array pointed to by **`s`**. No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

Returns

3 The **`fgets`** function returns **`s`** if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

7.21.7.3 The `fputc` function

Synopsis

1

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

Description

2 The **`fputc`** function writes the character specified by **`c`** (converted to an **`unsigned char`**) to the output stream pointed to by **`stream`**, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Returns

3 The **`fputc`** function returns the character written. If a write error occurs, the error indicator for the stream is set and **`fputc`** returns **`EOF`**.

7.21.7.4 The `fputs` function

Synopsis

1

```
#include <stdio.h>
int fputs(const char * restrict s,
     FILE * restrict stream);
```

Description

2 The **fputs** function writes the string pointed to by **s** to the stream pointed to by **stream**. The terminating null character is not written.

Returns

3 The **fputs** function returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

7.21.7.5 The getc function

Synopsis

1
```
#include <stdio.h>
int getc(FILE *stream);
```

Description

2 The **getc** function is equivalent to **fgetc**, except that if it is implemented as a macro, it may evaluate **stream** more than once, so the argument should never be an expression with side effects.

Returns

3 The **getc** function returns the next character from the input stream pointed to by **stream**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getc** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getc** returns **EOF**.

7.21.7.6 The getchar function

Synopsis

1
```
#include <stdio.h>
int getchar(void);
```

Description

2 The **getchar** function is equivalent to **getc** with the argument **stdin**.

Returns

3 The **getchar** function returns the next character from the input stream pointed to by **stdin**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getchar** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getchar** returns **EOF**.

7.21.7.7 The `putc` function

Synopsis

1
```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Description

2 The **putc** function is equivalent to **fputc**, except that if it is implemented as a macro, it may evaluate **stream** more than once, so that argument should never be an expression with side effects.

Returns

3 The **putc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **putc** returns **EOF**.

7.21.7.8 The `putchar` function

Synopsis

1
```
#include <stdio.h>
int putchar(int c);
```

Description

2 The **putchar** function is equivalent to **putc** with the second argument **stdout**.

Returns

3 The **putchar** function returns the character written. If a write error occurs, the error indicator for the stream is set and **putchar** returns **EOF**.

7.21.7.9 The `puts` function

Synopsis

1
```
#include <stdio.h>
int puts(const char *s);
```

Description

2 The **puts** function writes the string pointed to by **s** to the stream pointed to by **stdout**, and appends a new-line character to the output. The terminating null character is not written.

Returns

3 The **puts** function returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

7.21.7.10 The `ungetc` function

Synopsis

1
```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Description

2 The **`ungetc`** function pushes the character specified by **`c`** (converted to an **`unsigned char`**) back onto the input stream pointed to by **`stream`**. Pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by **`stream`**) to a file positioning function (**`fseek`**, **`fsetpos`**, or **`rewind`**) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

3 One character of pushback is guaranteed. If the **`ungetc`** function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

4 If the value of **`c`** equals that of the macro **`EOF`**, the operation fails and the input stream is unchanged.

5 A successful call to the **`ungetc`** function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back. For a text stream, the value of its file position indicator after a successful call to the **`ungetc`** function is unspecified until all pushed-back characters are read or discarded. For a binary stream, its file position indicator is decremented by each successful call to the **`ungetc`** function; if its value was zero before a call, it is indeterminate after the call.[290]

Returns

6 The **`ungetc`** function returns the character pushed back after conversion, or **`EOF`** if the operation fails.

Forward references: file positioning functions (7.21.9).

290) See ‘‘future library directions’’ (7.31.11).

7.21.8 Direct input/output functions

7.21.8.1 The `fread` function

Synopsis

1
```
#include <stdio.h>
size_t fread(void * restrict ptr,
     size_t size, size_t nmemb,
     FILE * restrict stream);
```

Description

2 The **`fread`** function reads, into the array pointed to by **`ptr`**, up to **`nmemb`** elements whose size is specified by **`size`**, from the stream pointed to by **`stream`**. For each object, **`size`** calls are made to the **`fgetc`** function and the results stored, in the order read, in an array of **`unsigned char`** exactly overlaying the object. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

Returns

3 The **`fread`** function returns the number of elements successfully read, which may be less than **`nmemb`** if a read error or end-of-file is encountered. If **`size`** or **`nmemb`** is zero, **`fread`** returns zero and the contents of the array and the state of the stream remain unchanged.

7.21.8.2 The `fwrite` function

Synopsis

1
```
#include <stdio.h>
size_t fwrite(const void * restrict ptr,
     size_t size, size_t nmemb,
     FILE * restrict stream);
```

Description

2 The **`fwrite`** function writes, from the array pointed to by **`ptr`**, up to **`nmemb`** elements whose size is specified by **`size`**, to the stream pointed to by **`stream`**. For each object, **`size`** calls are made to the **`fputc`** function, taking the values (in order) from an array of **`unsigned char`** exactly overlaying the object. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

Returns

3 The **fwrite** function returns the number of elements successfully written, which will be less than **nmemb** only if a write error is encountered. If **size** or **nmemb** is zero, **fwrite** returns zero and the state of the stream remains unchanged.

7.21.9 File positioning functions

7.21.9.1 The fgetpos function

Synopsis

1
```
#include <stdio.h>
int fgetpos(FILE * restrict stream,
     fpos_t * restrict pos);
```

Description

2 The **fgetpos** function stores the current values of the parse state (if any) and file position indicator for the stream pointed to by **stream** in the object pointed to by **pos**. The values stored contain unspecified information usable by the **fsetpos** function for repositioning the stream to its position at the time of the call to the **fgetpos** function.

Returns

3 If successful, the **fgetpos** function returns zero; on failure, the **fgetpos** function returns nonzero and stores an implementation-defined positive value in **errno**.

Forward references: the **fsetpos** function (7.21.9.3).

7.21.9.2 The fseek function

Synopsis

1
```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

Description

2 The **fseek** function sets the file position indicator for the stream pointed to by **stream**. If a read or write error occurs, the error indicator for the stream is set and **fseek** fails.

3 For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding **offset** to the position specified by **whence**. The specified position is the beginning of the file if **whence** is **SEEK_SET**, the current value of the file position indicator if **SEEK_CUR**, or end-of-file if **SEEK_END**. A binary stream need not meaningfully support **fseek** calls with a **whence** value of **SEEK_END**.

4 For a text stream, either **offset** shall be zero, or **offset** shall be a value returned by an earlier successful call to the **ftell** function on a stream associated with the same file and **whence** shall be **SEEK_SET**.

5 After determining the new position, a successful call to the `fseek` function undoes any effects of the `ungetc` function on the stream, clears the end-of-file indicator for the stream, and then establishes the new position. After a successful `fseek` call, the next operation on an update stream may be either input or output.

Returns

6 The `fseek` function returns nonzero only for a request that cannot be satisfied.

Forward references: the `ftell` function (7.21.9.4).

7.21.9.3 The `fsetpos` function

Synopsis

1
```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

2 The `fsetpos` function sets the `mbstate_t` object (if any) and file position indicator for the stream pointed to by `stream` according to the value of the object pointed to by `pos`, which shall be a value obtained from an earlier successful call to the `fgetpos` function on a stream associated with the same file. If a read or write error occurs, the error indicator for the stream is set and `fsetpos` fails.

3 A successful call to the `fsetpos` function undoes any effects of the `ungetc` function on the stream, clears the end-of-file indicator for the stream, and then establishes the new parse state and position. After a successful `fsetpos` call, the next operation on an update stream may be either input or output.

Returns

4 If successful, the `fsetpos` function returns zero; on failure, the `fsetpos` function returns nonzero and stores an implementation-defined positive value in `errno`.

7.21.9.4 The `ftell` function

Synopsis

1
```
#include <stdio.h>
long int ftell(FILE *stream);
```

Description

2 The `ftell` function obtains the current value of the file position indicator for the stream pointed to by `stream`. For a binary stream, the value is the number of characters from the beginning of the file. For a text stream, its file position indicator contains unspecified information, usable by the `fseek` function for returning the file position indicator for the stream to its position at the time of the `ftell` call; the difference between two such return values is not necessarily a meaningful measure of the number of characters written

or read.

Returns

3 If successful, the **`ftell`** function returns the current value of the file position indicator for the stream. On failure, the **`ftell`** function returns −1L and stores an implementation-defined positive value in **`errno`**.

7.21.9.5 The `rewind` function

Synopsis

1
```
#include <stdio.h>
void rewind(FILE *stream);
```

Description

2 The **`rewind`** function sets the file position indicator for the stream pointed to by **`stream`** to the beginning of the file. It is equivalent to

```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared.

Returns

3 The **`rewind`** function returns no value.

7.21.10 Error-handling functions

7.21.10.1 The `clearerr` function

Synopsis

1
```
#include <stdio.h>
void clearerr(FILE *stream);
```

Description

2 The **`clearerr`** function clears the end-of-file and error indicators for the stream pointed to by **`stream`**.

Returns

3 The **`clearerr`** function returns no value.

7.21.10.2 The `feof` function

Synopsis

1
```
#include <stdio.h>
int feof(FILE *stream);
```

Description

2 The **`feof`** function tests the end-of-file indicator for the stream pointed to by **`stream`**.

Returns

3 The **`feof`** function returns nonzero if and only if the end-of-file indicator is set for **`stream`**.

7.21.10.3 The `ferror` function

Synopsis

1
```
#include <stdio.h>
int ferror(FILE *stream);
```

Description

2 The **`ferror`** function tests the error indicator for the stream pointed to by **`stream`**.

Returns

3 The **`ferror`** function returns nonzero if and only if the error indicator is set for **`stream`**.

7.21.10.4 The `perror` function

Synopsis

1
```
#include <stdio.h>
void perror(const char *s);
```

Description

2 The **`perror`** function maps the error number in the integer expression **`errno`** to an error message. It writes a sequence of characters to the standard error stream thus: first (if **`s`** is not a null pointer and the character pointed to by **`s`** is not the null character), the string pointed to by **`s`** followed by a colon (**`:`**) and a space; then an appropriate error message string followed by a new-line character. The contents of the error message strings are the same as those returned by the **`strerror`** function with argument **`errno`**.

Returns

3 The **`perror`** function returns no value.

Forward references: the **`strerror`** function (7.24.6.2).

7.22 General utilities `<stdlib.h>`

1 The header **`<stdlib.h>`** declares five types and several functions of general utility, and defines several macros.[291)]

2 The types declared are **`size_t`** and **`wchar_t`** (both described in 7.19),

`div_t`

which is a structure type that is the type of the value returned by the **`div`** function,

`ldiv_t`

which is a structure type that is the type of the value returned by the **`ldiv`** function, and

`lldiv_t`

which is a structure type that is the type of the value returned by the **`lldiv`** function.

3 The macros defined are **`NULL`** (described in 7.19);

`EXIT_FAILURE`

and

`EXIT_SUCCESS`

which expand to integer constant expressions that can be used as the argument to the **`exit`** function to return unsuccessful or successful termination status, respectively, to the host environment;

`RAND_MAX`

which expands to an integer constant expression that is the maximum value returned by the **`rand`** function; and

`MB_CUR_MAX`

which expands to a positive integer expression with type **`size_t`** that is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category **`LC_CTYPE`**), which is never greater than **`MB_LEN_MAX`**.

291) See ‘‘future library directions’’ (7.31.12).

7.22.1 Numeric conversion functions

1 The functions **atof**, **atoi**, **atol**, and **atoll** need not affect the value of the integer expression **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

7.22.1.1 The atof function

Synopsis

1
```
#include <stdlib.h>
double atof(const char *nptr);
```

Description

2 The **atof** function converts the initial portion of the string pointed to by **nptr** to **double** representation. Except for the behavior on error, it is equivalent to

```
strtod(nptr, (char **)NULL)
```

Returns

3 The **atof** function returns the converted value.

Forward references: the **strtod**, **strtof**, and **strtold** functions (7.22.1.3).

7.22.1.2 The atoi, atol, and atoll functions

Synopsis

1
```
#include <stdlib.h>
int atoi(const char *nptr);
long int atol(const char *nptr);
long long int atoll(const char *nptr);
```

Description

2 The **atoi**, **atol**, and **atoll** functions convert the initial portion of the string pointed to by **nptr** to **int**, **long int**, and **long long int** representation, respectively. Except for the behavior on error, they are equivalent to

```
atoi:  (int)strtol(nptr, (char **)NULL, 10)
atol:  strtol(nptr, (char **)NULL, 10)
atoll: strtoll(nptr, (char **)NULL, 10)
```

Returns

3 The **atoi**, **atol**, and **atoll** functions return the converted value.

Forward references: the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.22.1.4).

7.22.1.3 The `strtod`, `strtof`, and `strtold` functions

Synopsis

1
```
#include <stdlib.h>
double strtod(const char * restrict nptr,
     char ** restrict endptr);
float strtof(const char * restrict nptr,
     char ** restrict endptr);
long double strtold(const char * restrict nptr,
     char ** restrict endptr);
```

Description

2 The **`strtod`**, **`strtof`**, and **`strtold`** functions convert the initial portion of the string pointed to by **`nptr`** to **`double`**, **`float`**, and **`long double`** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **`isspace`** function), a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

3 The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

— a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part as defined in 6.4.4.2;

— a **`0x`** or **`0X`**, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary exponent part as defined in 6.4.4.2;

— **`INF`** or **`INFINITY`**, ignoring case

— **`NAN`** or **`NAN(`***n-char-sequence*$_{opt}$**`)`**, ignoring case in the **`NAN`** part, where:

> *n-char-sequence:*
> *digit*
> *nondigit*
> *n-char-sequence digit*
> *n-char-sequence nondigit*

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

4 If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, except that the

decimal-point character is used in place of a period, and that if neither an exponent part nor a decimal-point character appears in a decimal floating point number, or if a binary exponent part does not appear in a hexadecimal floating point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated.[292)] A character sequence **`INF`** or **`INFINITY`** is interpreted as an infinity, if representable in the return type, else like a floating constant that is too large for the range of the return type. A character sequence **`NAN`** or **`NAN(`***n-char-sequence*$_{opt}$**`)`** is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the meaning of the n-char sequence is implementation-defined.[293)] A pointer to the final string is stored in the object pointed to by **`endptr`**, provided that **`endptr`** is not a null pointer.

5 If the subject sequence has the hexadecimal form and **`FLT_RADIX`** is a power of 2, the value resulting from the conversion is correctly rounded.

6 In other than the **`"C"`** locale, additional locale-specific subject sequence forms may be accepted.

7 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **`nptr`** is stored in the object pointed to by **`endptr`**, provided that **`endptr`** is not a null pointer.

Recommended practice

8 If the subject sequence has the hexadecimal form, **`FLT_RADIX`** is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

9 If the subject sequence has the decimal form and at most **`DECIMAL_DIG`** (defined in **`<float.h>`**) significant digits, the result should be correctly rounded. If the subject sequence D has the decimal form and more than **`DECIMAL_DIG`** significant digits, consider the two bounding, adjacent decimal strings L and U, both having **`DECIMAL_DIG`** significant digits, such that the values of L, D, and U satisfy $L \le D \le U$. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra

292) It is unspecified whether a minus-signed sequence is converted to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (see F.5); the two methods may yield different results if rounding is toward positive or negative infinity. In either case, the functions honor the sign of zero if floating-point arithmetic supports signed zeros.

293) An implementation may use the n-char sequence to determine extra information to be represented in the NaN's significand.

stipulation that the error with respect to *D* should have a correct sign for the current rounding direction.[294)]

Returns

10 The functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value overflows and default rounding is in effect (7.12.1), plus or minus **`HUGE_VAL`**, **`HUGE_VALF`**, or **`HUGE_VALL`** is returned (according to the return type and sign of the value), and the value of the macro **`ERANGE`** is stored in **`errno`**. If the result underflows (7.12.1), the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether **`errno`** acquires the value **`ERANGE`** is implementation-defined.

7.22.1.4 The `strtol`, `strtoll`, `strtoul`, and `strtoull` functions

Synopsis

1
```
#include <stdlib.h>
long int strtol(
     const char * restrict nptr,
     char ** restrict endptr,
     int base);
long long int strtoll(
     const char * restrict nptr,
     char ** restrict endptr,
     int base);
unsigned long int strtoul(
     const char * restrict nptr,
     char ** restrict endptr,
     int base);
unsigned long long int strtoull(
     const char * restrict nptr,
     char ** restrict endptr,
     int base);
```

Description

2 The **`strtol`**, **`strtoll`**, **`strtoul`**, and **`strtoull`** functions convert the initial portion of the string pointed to by **`nptr`** to **`long int`**, **`long long int`**, **`unsigned long int`**, and **`unsigned long long int`** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **`isspace`** function), a subject sequence

294) **`DECIMAL_DIG`**, defined in **`<float.h>`**, should be sufficiently large that *L* and *U* will usually round to the same internal floating value, but if not will round to adjacent values.

resembling an integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to an integer, and return the result.

3 If the value of **base** is zero, the expected form of the subject sequence is that of an integer constant as described in 6.4.4.1, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by **base**, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from **a** (or **A**) through **z** (or **Z**) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted. If the value of **base** is 16, the characters **0x** or **0X** may optionally precede the sequence of letters and digits, following the sign if present.

4 The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

5 If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to the rules of 6.4.4.1. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated (in the return type). A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

6 In other than the **"C"** locale, additional locale-specific subject sequence forms may be accepted.

7 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Returns

8 The **strtol**, **strtoll**, **strtoul**, and **strtoull** functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG_MIN**, **LONG_MAX**, **LLONG_MIN**, **LLONG_MAX**, **ULONG_MAX**, or **ULLONG_MAX** is returned (according to the return type and sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

7.22.2 Pseudo-random sequence generation functions

7.22.2.1 The `rand` function

Synopsis

1
```
#include <stdlib.h>
int rand(void);
```

Description

2 The **`rand`** function computes a sequence of pseudo-random integers in the range 0 to **`RAND_MAX`**.[295)]

3 The **`rand`** function is not required to avoid data races with other calls to pseudo-random sequence generation functions. The implementation shall behave as if no library function calls the **`rand`** function.

Returns

4 The **`rand`** function returns a pseudo-random integer.

Environmental limits

5 The value of the **`RAND_MAX`** macro shall be at least 32767.

7.22.2.2 The `srand` function

Synopsis

1
```
#include <stdlib.h>
void srand(unsigned int seed);
```

Description

2 The **`srand`** function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **`rand`**. If **`srand`** is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If **`rand`** is called before any calls to **`srand`** have been made, the same sequence shall be generated as when **`srand`** is first called with a seed value of 1.

3 The **`srand`** function is not required to avoid data races with other calls to pseudo-random sequence generation functions. The implementation shall behave as if no library function calls the **`srand`** function.

---

295) There are no guarantees as to the quality of the random sequence produced and some implementations are known to produce sequences with distressingly non-random low-order bits. Applications with particular requirements should use a generator that is known to be sufficient for their needs.

Returns

4 The **`srand`** function returns no value.

5 EXAMPLE The following functions define a portable implementation of **`rand`** and **`srand`**.

```
static unsigned long int next = 1;

int rand(void)   // RAND_MAX assumed to be 32767
{
      next = next * 1103515245 + 12345;
      return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
      next = seed;
}
```

7.22.3 Memory management functions

1 The order and contiguity of storage allocated by successive calls to the **`aligned_alloc`**, **`calloc`**, **`malloc`**, and **`realloc`** functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated). The lifetime of an allocated object extends from the allocation until the deallocation. Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.

2 For purposes of determining the existence of a data race, memory allocation functions behave as though they accessed only memory locations accessible through their arguments and not other static duration storage. These functions may, however, visibly modify the storage that they allocate or deallocate. A call to **`free`** or **`realloc`** that deallocates a region *p* of memory synchronizes with any allocation call that allocates all or part of the region *p*. This synchronization occurs after any access of *p* by the deallocating function, and before any such access by the allocating function.

7.22.3.1 The `aligned_alloc` function

Synopsis

1
```
#include <stdlib.h>
void *aligned_alloc(size_t alignment, size_t size);
```

Description

2 The **aligned_alloc** function allocates space for an object whose alignment is specified by **alignment**, whose size is specified by **size**, and whose value is indeterminate. The value of **alignment** shall be a valid alignment supported by the implementation and the value of **size** shall be an integral multiple of **alignment**.

Returns

3 The **aligned_alloc** function returns either a null pointer or a pointer to the allocated space.

7.22.3.2 The calloc function

Synopsis

1
```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Description

2 The **calloc** function allocates space for an array of **nmemb** objects, each of whose size is **size**. The space is initialized to all bits zero.[296)]

Returns

3 The **calloc** function returns either a null pointer or a pointer to the allocated space.

7.22.3.3 The free function

Synopsis

1
```
#include <stdlib.h>
void free(void *ptr);
```

Description

2 The **free** function causes the space pointed to by **ptr** to be deallocated, that is, made available for further allocation. If **ptr** is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by a memory management function, or if the space has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

Returns

3 The **free** function returns no value.

---

296) Note that this need not be the same as the representation of floating-point zero or a null pointer constant.

7.22.3.4 The `malloc` function

Synopsis

1
```
#include <stdlib.h>
void *malloc(size_t size);
```

Description

2 The **`malloc`** function allocates space for an object whose size is specified by **`size`** and whose value is indeterminate.

Returns

3 The **`malloc`** function returns either a null pointer or a pointer to the allocated space.

7.22.3.5 The `realloc` function

Synopsis

1
```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

Description

2 The **`realloc`** function deallocates the old object pointed to by **`ptr`** and returns a pointer to a new object that has the size specified by **`size`**. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

3 If **`ptr`** is a null pointer, the **`realloc`** function behaves like the **`malloc`** function for the specified size. Otherwise, if **`ptr`** does not match a pointer earlier returned by a memory management function, or if the space has been deallocated by a call to the **`free`** or **`realloc`** function, the behavior is undefined. If memory for the new object cannot be allocated, the old object is not deallocated and its value is unchanged.

Returns

4 The **`realloc`** function returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object could not be allocated.

7.22.4 Communication with the environment

7.22.4.1 The `abort` function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void abort(void);
```

Description

2 The **`abort`** function causes abnormal program termination to occur, unless the signal **`SIGABRT`** is being caught and the signal handler does not return. Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined. An implementation-defined form of the status *unsuccessful termination* is returned to the host environment by means of the function call **`raise(SIGABRT)`**.

Returns

3 The **`abort`** function does not return to its caller.

7.22.4.2 The `atexit` function

Synopsis

1
```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Description

2 The **`atexit`** function registers the function pointed to by **`func`**, to be called without arguments at normal program termination.[297)] It is unspecified whether a call to the **`atexit`** function that does not happen before the **`exit`** function is called will succeed.

Environmental limits

3 The implementation shall support the registration of at least 32 functions.

Returns

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

Forward references: the **`at_quick_exit`** function (7.22.4.3), the **`exit`** function (7.22.4.4).

297) The **`atexit`** function registrations are distinct from the **`at_quick_exit`** registrations, so applications may need to call both registration functions with the same argument.

7.22.4.3 The `at_quick_exit` function

Synopsis

1
```
#include <stdlib.h>
int at_quick_exit(void (*func)(void));
```

Description

2 The **at_quick_exit** function registers the function pointed to by **func**, to be called without arguments should **quick_exit** be called.[298] It is unspecified whether a call to the **at_quick_exit** function that does not happen before the **quick_exit** function is called will succeed.

Environmental limits

3 The implementation shall support the registration of at least 32 functions.

Returns

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

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

7.22.4.4 The `exit` function

Synopsis

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

Description

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

3 First, all functions registered by the **atexit** function are called, in the reverse order of their registration,[299] except that a function is called after any previously registered functions that had already been called at the time it was registered. If, during the call to any such function, a call to the **longjmp** function is made that would terminate the call to the registered function, the behavior is undefined.

---

298) The **at_quick_exit** function registrations are distinct from the **atexit** registrations, so applications may need to call both registration functions with the same argument.

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

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

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

Returns

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

7.22.4.5 The _Exit function

Synopsis

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

Description

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

Returns

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

7.22.4.6 The getenv function

Synopsis

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

Description

2 The **getenv** function searches an *environment list*, provided by the host environment, for a string that matches the string pointed to by **name**. The set of environment names and the method for altering the environment list are implementation-defined. The **getenv** function need not avoid data races with other threads of execution that modify the environment list.[300)]

300) Many implementations provide non-standard functions that modify the environment list.

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

Returns

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

7.22.4.7 The quick_exit function

Synopsis

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

Description

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

3 The **quick_exit** function first calls all functions registered by the **at_quick_exit** function, in the reverse order of their registration,[301] except that a function is called after any previously registered functions that had already been called at the time it was registered. If, during the call to any such function, a call to the **longjmp** function is made that would terminate the call to the registered function, the behavior is undefined.

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

Returns

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

7.22.4.8 The system function

Synopsis

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

Description

2 If **string** is a null pointer, the **system** function determines whether the host environment has a *command processor*. If **string** is not a null pointer, the **system**

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

function passes the string pointed to by **string** to that command processor to be executed in a manner which the implementation shall document; this might then cause the program calling **system** to behave in a non-conforming manner or to terminate.

Returns

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

7.22.5 Searching and sorting utilities

1 These utilities make use of a comparison function to search or sort arrays of unspecified type. Where an argument declared as **size_t nmemb** specifies the length of the array for a function, **nmemb** can have the value zero on a call to that function; the comparison function is not called, a search finds no matching element, and sorting performs no rearrangement. Pointer arguments on such a call shall still have valid values, as described in 7.1.4.

2 The implementation shall ensure that the second argument of the comparison function (when called from **bsearch**), or both arguments (when called from **qsort**), are pointers to elements of the array.[302] The first argument when called from **bsearch** shall equal **key**.

3 The comparison function shall not alter the contents of the array. The implementation may reorder elements of the array between calls to the comparison function, but shall not alter the contents of any individual element.

4 When the same objects (consisting of **size** bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, for **qsort** they shall define a total ordering on the array, and for **bsearch** the same object shall always compare the same way with the key.

5 A sequence point occurs immediately before and immediately after each call to the comparison function, and also between any call to the comparison function and any movement of the objects passed as arguments to that call.

---

302) That is, if the value passed is **p**, then the following expressions are always nonzero:

```
((char *)p - (char *)base) % size == 0
(char *)p >= (char *)base
(char *)p < (char *)base + nmemb * size
```

7.22.5.1 The `bsearch` function

Synopsis

1
```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
     size_t nmemb, size_t size,
     int (*compar)(const void *, const void *));
```

Description

2 The **bsearch** function searches an array of **nmemb** objects, the initial element of which is pointed to by **base**, for an element that matches the object pointed to by **key**. The size of each element of the array is specified by **size**.

3 The comparison function pointed to by **compar** is called with two arguments that point to the **key** object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the **key** object is considered, respectively, to be less than, to match, or to be greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the **key** object, in that order.[303)]

Returns

4 The **bsearch** function returns a pointer to a matching element of the array, or a null pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

7.22.5.2 The `qsort` function

Synopsis

1
```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
     int (*compar)(const void *, const void *));
```

Description

2 The **qsort** function sorts an array of **nmemb** objects, the initial element of which is pointed to by **base**. The size of each object is specified by **size**.

3 The contents of the array are sorted into ascending order according to a comparison function pointed to by **compar**, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

303) In practice, the entire array is sorted according to the comparison function.

4 If two elements compare as equal, their order in the resulting sorted array is unspecified.

Returns

5 The **qsort** function returns no value.

7.22.6 Integer arithmetic functions

7.22.6.1 The abs, labs and llabs functions

Synopsis

1
```
#include <stdlib.h>
int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
```

Description

2 The **abs**, **labs**, and **llabs** functions compute the absolute value of an integer **j**. If the result cannot be represented, the behavior is undefined.[304)]

Returns

3 The **abs**, **labs**, and **llabs**, functions return the absolute value.

7.22.6.2 The div, ldiv, and lldiv functions

Synopsis

1
```
#include <stdlib.h>
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
```

Description

2 The **div**, **ldiv**, and **lldiv**, functions compute **numer / denom** and **numer % denom** in a single operation.

Returns

3 The **div**, **ldiv**, and **lldiv** functions return a structure of type **div_t**, **ldiv_t**, and **lldiv_t**, respectively, comprising both the quotient and the remainder. The structures shall contain (in either order) the members **quot** (the quotient) and **rem** (the remainder), each of which has the same type as the arguments **numer** and **denom**. If either part of the result cannot be represented, the behavior is undefined.

304) The absolute value of the most negative number cannot be represented in two's complement.

7.22.7 Multibyte/wide character conversion functions

1 The behavior of the multibyte character functions is affected by the **LC_CTYPE** category of the current locale. For a state-dependent encoding, each function is placed into its initial conversion state at program startup and can be returned to that state by a call for which its character pointer argument, **s**, is a null pointer. Subsequent calls with **s** as other than a null pointer cause the internal conversion state of the function to be altered as necessary. A call with **s** as a null pointer causes these functions to return a nonzero value if encodings have state dependency, and zero otherwise.[305)] Changing the **LC_CTYPE** category causes the conversion state of these functions to be indeterminate.

7.22.7.1 The mblen function

Synopsis

1
```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Description

2 If **s** is not a null pointer, the **mblen** function determines the number of bytes contained in the multibyte character pointed to by **s**. Except that the conversion state of the **mbtowc** function is not affected, it is equivalent to

```
mbtowc((wchar_t *)0, (const char *)0, 0);
mbtowc((wchar_t *)0, s, n);
```

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

Returns

4 If **s** is a null pointer, the **mblen** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If **s** is not a null pointer, the **mblen** function either returns 0 (if **s** points to the null character), or returns the number of bytes that are contained in the multibyte character (if the next **n** or fewer bytes form a valid multibyte character), or returns −1 (if they do not form a valid multibyte character).

Forward references: the **mbtowc** function (7.22.7.2).

305) If the locale employs special bytes to change the shift state, these bytes do not produce separate wide character codes, but are grouped with an adjacent multibyte character.

7.22.7.2 The `mbtowc` function

Synopsis

1
```
#include <stdlib.h>
int mbtowc(wchar_t * restrict pwc,
     const char * restrict s,
     size_t n);
```

Description

2 If **s** is not a null pointer, the **mbtowc** function inspects at most **n** bytes beginning with the byte pointed to by **s** to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the value of the corresponding wide character and then, if **pwc** is not a null pointer, stores that value in the object pointed to by **pwc**. If the corresponding wide character is the null wide character, the function is left in the initial conversion state.

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

Returns

4 If **s** is a null pointer, the **mbtowc** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If **s** is not a null pointer, the **mbtowc** function either returns 0 (if **s** points to the null character), or returns the number of bytes that are contained in the converted multibyte character (if the next **n** or fewer bytes form a valid multibyte character), or returns −1 (if they do not form a valid multibyte character).

5 In no case will the value returned be greater than **n** or the value of the **MB_CUR_MAX** macro.

7.22.7.3 The `wctomb` function

Synopsis

1
```
#include <stdlib.h>
int wctomb(char *s, wchar_t wc);
```

Description

2 The **wctomb** function determines the number of bytes needed to represent the multibyte character corresponding to the wide character given by **wc** (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by **s** (if **s** is not a null pointer). At most **MB_CUR_MAX** characters are stored. If **wc** is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state, and the function is left in the initial conversion state.

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

Returns

4 If **s** is a null pointer, the **wctomb** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If **s** is not a null pointer, the **wctomb** function returns −1 if the value of **wc** does not correspond to a valid multibyte character, or returns the number of bytes that are contained in the multibyte character corresponding to the value of **wc**.

5 In no case will the value returned be greater than the value of the **MB_CUR_MAX** macro.

7.22.8 Multibyte/wide string conversion functions

1 The behavior of the multibyte string functions is affected by the **LC_CTYPE** category of the current locale.

7.22.8.1 The mbstowcs function

Synopsis

1
```
#include <stdlib.h>
size_t mbstowcs(wchar_t * restrict pwcs,
     const char * restrict s,
     size_t n);
```

Description

2 The **mbstowcs** function converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by **s** into a sequence of corresponding wide characters and stores not more than **n** wide characters into the array pointed to by **pwcs**. No multibyte characters that follow a null character (which is converted into a null wide character) will be examined or converted. Each multibyte character is converted as if by a call to the **mbtowc** function, except that the conversion state of the **mbtowc** function is not affected.

3 No more than **n** elements will be modified in the array pointed to by **pwcs**. If copying takes place between objects that overlap, the behavior is undefined.

Returns

4 If an invalid multibyte character is encountered, the **mbstowcs** function returns **(size_t)(-1)**. Otherwise, the **mbstowcs** function returns the number of array elements modified, not including a terminating null wide character, if any.[306)]

306) The array will not be null-terminated if the value returned is **n**.

7.22.8.2 The `wcstombs` function

Synopsis

1
```
#include <stdlib.h>
size_t wcstombs(char * restrict s,
     const wchar_t * restrict pwcs,
     size_t n);
```

Description

2 The **wcstombs** function converts a sequence of wide characters from the array pointed to by **pwcs** into a sequence of corresponding multibyte characters that begins in the initial shift state, and stores these multibyte characters into the array pointed to by **s**, stopping if a multibyte character would exceed the limit of **n** total bytes or if a null character is stored. Each wide character is converted as if by a call to the **wctomb** function, except that the conversion state of the **wctomb** function is not affected.

3 No more than **n** bytes will be modified in the array pointed to by **s**. If copying takes place between objects that overlap, the behavior is undefined.

Returns

4 If a wide character is encountered that does not correspond to a valid multibyte character, the **wcstombs** function returns **(size_t)(-1)**. Otherwise, the **wcstombs** function returns the number of bytes modified, not including a terminating null character, if any.[306)]

7.23 `_Noreturn` `<stdnoreturn.h>`

1 The header **`<stdnoreturn.h>`** defines the macro

`noreturn`

which expands to **`_Noreturn`**.

7.24 String handling <string.h>

7.24.1 String function conventions

1 The header **<string.h>** declares one type and several functions, and defines one macro useful for manipulating arrays of character type and other objects treated as arrays of character type.[307)] The type is **size_t** and the macro is **NULL** (both described in 7.19). Various methods are used for determining the lengths of the arrays, but in all cases a **char *** or **void *** argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

2 Where an argument declared as **size_t n** specifies the length of the array for a function, **n** can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid values, as described in 7.1.4. On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

3 For all functions in this subclause, each character shall be interpreted as if it had the type **unsigned char** (and therefore every possible object representation is valid and has a different value).

7.24.2 Copying functions

7.24.2.1 The memcpy function

Synopsis

1
```
#include <string.h>
void *memcpy(void * restrict s1,
     const void * restrict s2,
     size_t n);
```

Description

2 The **memcpy** function copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **memcpy** function returns the value of **s1**.

307) See ‘‘future library directions’’ (7.31.13).

7.24.2.2 The `memmove` function

Synopsis

1
```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Description

2 The **memmove** function copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. Copying takes place as if the **n** characters from the object pointed to by **s2** are first copied into a temporary array of **n** characters that does not overlap the objects pointed to by **s1** and **s2**, and then the **n** characters from the temporary array are copied into the object pointed to by **s1**.

Returns

3 The **memmove** function returns the value of **s1**.

7.24.2.3 The `strcpy` function

Synopsis

1
```
#include <string.h>
char *strcpy(char * restrict s1,
     const char * restrict s2);
```

Description

2 The **strcpy** function copies the string pointed to by **s2** (including the terminating null character) into the array pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **strcpy** function returns the value of **s1**.

7.24.2.4 The `strncpy` function

Synopsis

1
```
#include <string.h>
char *strncpy(char * restrict s1,
     const char * restrict s2,
     size_t n);
```

Description

2 The **strncpy** function copies not more than **n** characters (characters that follow a null character are not copied) from the array pointed to by **s2** to the array pointed to by

s1.[308] If copying takes place between objects that overlap, the behavior is undefined.

3 If the array pointed to by **s2** is a string that is shorter than **n** characters, null characters are appended to the copy in the array pointed to by **s1**, until **n** characters in all have been written.

Returns

4 The **strncpy** function returns the value of **s1**.

7.24.3 Concatenation functions

7.24.3.1 The strcat function

Synopsis

1
```
#include <string.h>
char *strcat(char * restrict s1,
     const char * restrict s2);
```

Description

2 The **strcat** function appends a copy of the string pointed to by **s2** (including the terminating null character) to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **strcat** function returns the value of **s1**.

7.24.3.2 The strncat function

Synopsis

1
```
#include <string.h>
char *strncat(char * restrict s1,
     const char * restrict s2,
     size_t n);
```

Description

2 The **strncat** function appends not more than **n** characters (a null character and characters that follow it are not appended) from the array pointed to by **s2** to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**. A terminating null character is always appended to the result.[309] If copying

308) Thus, if there is no null character in the first **n** characters of the array pointed to by **s2**, the result will not be null-terminated.

309) Thus, the maximum number of characters that can end up in the array pointed to by **s1** is **strlen(s1)+n+1**.

takes place between objects that overlap, the behavior is undefined.

Returns

3 The **strncat** function returns the value of **s1**.

Forward references: the **strlen** function (7.24.6.3).

7.24.4 Comparison functions

1 The sign of a nonzero value returned by the comparison functions **memcmp**, **strcmp**, and **strncmp** is determined by the sign of the difference between the values of the first pair of characters (both interpreted as **unsigned char**) that differ in the objects being compared.

7.24.4.1 The memcmp function

Synopsis

1
```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Description

2 The **memcmp** function compares the first **n** characters of the object pointed to by **s1** to the first **n** characters of the object pointed to by **s2**.[310)]

Returns

3 The **memcmp** function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by **s1** is greater than, equal to, or less than the object pointed to by **s2**.

7.24.4.2 The strcmp function

Synopsis

1
```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Description

2 The **strcmp** function compares the string pointed to by **s1** to the string pointed to by **s2**.

Returns

3 The **strcmp** function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by **s1** is greater than, equal to, or less than the string

310) The contents of ‘‘holes’’ used as padding for purposes of alignment within structure objects are indeterminate. Strings shorter than their allocated space and unions may also cause problems in comparison.

pointed to by `s2`.

7.24.4.3 The `strcoll` function

Synopsis

1
```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Description

2 The `strcoll` function compares the string pointed to by `s1` to the string pointed to by `s2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale.

Returns

3 The `strcoll` function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2` when both are interpreted as appropriate to the current locale.

7.24.4.4 The `strncmp` function

Synopsis

1
```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

Description

2 The `strncmp` function compares not more than `n` characters (characters that follow a null character are not compared) from the array pointed to by `s1` to the array pointed to by `s2`.

Returns

3 The `strncmp` function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by `s1` is greater than, equal to, or less than the possibly null-terminated array pointed to by `s2`.

7.24.4.5 The `strxfrm` function

Synopsis

1
```
#include <string.h>
size_t strxfrm(char * restrict s1,
     const char * restrict s2,
     size_t n);
```

Description

2 The `strxfrm` function transforms the string pointed to by `s2` and places the resulting string into the array pointed to by `s1`. The transformation is such that if the `strcmp` function is applied to two transformed strings, it returns a value greater than, equal to, or

less than zero, corresponding to the result of the **strcoll** function applied to the same two original strings. No more than **n** characters are placed into the resulting array pointed to by **s1**, including the terminating null character. If **n** is zero, **s1** is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **strxfrm** function returns the length of the transformed string (not including the terminating null character). If the value returned is **n** or more, the contents of the array pointed to by **s1** are indeterminate.

4 EXAMPLE The value of the following expression is the size of the array needed to hold the transformation of the string pointed to by **s**.

```
1 + strxfrm(NULL, s, 0)
```

7.24.5 Search functions

7.24.5.1 The memchr function

Synopsis

1
```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

Description

2 The **memchr** function locates the first occurrence of **c** (converted to an **unsigned char**) in the initial **n** characters (each interpreted as **unsigned char**) of the object pointed to by **s**. The implementation shall behave as if it reads the characters sequentially and stops as soon as a matching character is found.

Returns

3 The **memchr** function returns a pointer to the located character, or a null pointer if the character does not occur in the object.

7.24.5.2 The strchr function

Synopsis

1
```
#include <string.h>
char *strchr(const char *s, int c);
```

Description

2 The **strchr** function locates the first occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

Returns

3 The **strchr** function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

7.24.5.3 The strcspn function

Synopsis

1
```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

Description

2 The **strcspn** function computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters *not* from the string pointed to by **s2**.

Returns

3 The **strcspn** function returns the length of the segment.

7.24.5.4 The strpbrk function

Synopsis

1
```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

Description

2 The **strpbrk** function locates the first occurrence in the string pointed to by **s1** of any character from the string pointed to by **s2**.

Returns

3 The **strpbrk** function returns a pointer to the character, or a null pointer if no character from **s2** occurs in **s1**.

7.24.5.5 The strrchr function

Synopsis

1
```
#include <string.h>
char *strrchr(const char *s, int c);
```

Description

2 The **strrchr** function locates the last occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

Returns

3 The **strrchr** function returns a pointer to the character, or a null pointer if **c** does not occur in the string.

7.24.5.6 The strspn function

Synopsis

1
```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Description

2 The **strspn** function computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

Returns

3 The **strspn** function returns the length of the segment.

7.24.5.7 The strstr function

Synopsis

1
```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Description

2 The **strstr** function locates the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2**.

Returns

3 The **strstr** function returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, the function returns **s1**.

7.24.5.8 The strtok function

Synopsis

1
```
#include <string.h>
char *strtok(char * restrict s1,
     const char * restrict s2);
```

Description

2 A sequence of calls to the **strtok** function breaks the string pointed to by **s1** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **s2**. The first call in the sequence has a non-null first argument; subsequent calls in the sequence have a null first argument. The separator string pointed to by **s2** may be different from call to call.

3 The first call in the sequence searches the string pointed to by **s1** for the first character that is *not* contained in the current separator string pointed to by **s2**. If no such character is found, then there are no tokens in the string pointed to by **s1** and the **strtok** function returns a null pointer. If such a character is found, it is the start of the first token.

4 The **strtok** function then searches from there for a character that *is* contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **s1**, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. The **strtok** function saves a pointer to the following character, from which the next search for a token will start.

5 Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

6 The **strtok** function is not required to avoid data races with other calls to the **strtok** function.[311] The implementation shall behave as if no library function calls the **strtok** function.

Returns

7 The **strtok** function returns a pointer to the first character of a token, or a null pointer if there is no token.

8 EXAMPLE

```
#include <string.h>
static char str[] = "?a???b,,,#c";
char *t;

t = strtok(str, "?");    // t points to the token "a"
t = strtok(NULL, ",");   // t points to the token "??b"
t = strtok(NULL, "#,");  // t points to the token "c"
t = strtok(NULL, "?");   // t is a null pointer
```

Forward references: The **strtok_s** function (K.3.7.3.1).

311) The **strtok_s** function can be used instead to avoid data races.

7.24.6 Miscellaneous functions

7.24.6.1 The `memset` function

Synopsis

1
```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

Description

2 The `memset` function copies the value of `c` (converted to an `unsigned char`) into each of the first `n` characters of the object pointed to by `s`.

Returns

3 The `memset` function returns the value of `s`.

7.24.6.2 The `strerror` function

Synopsis

1
```
#include <string.h>
char *strerror(int errnum);
```

Description

2 The `strerror` function maps the number in `errnum` to a message string. Typically, the values for `errnum` come from `errno`, but `strerror` shall map any value of type `int` to a message.

3 The `strerror` function is not required to avoid data races with other calls to the `strerror` function.[312] The implementation shall behave as if no library function calls the `strerror` function.

Returns

4 The `strerror` function returns a pointer to the string, the contents of which are locale-specific. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `strerror` function.

Forward references: The `strerror_s` function (K.3.7.4.2).

312) The `strerror_s` function can be used instead to avoid data races.

7.24.6.3 The `strlen` function

Synopsis

1
```
#include <string.h>
size_t strlen(const char *s);
```

Description

2 The **`strlen`** function computes the length of the string pointed to by **`s`**.

Returns

3 The **`strlen`** function returns the number of characters that precede the terminating null character.

7.25 Type-generic math `<tgmath.h>`

1 The header `<tgmath.h>` includes the headers `<math.h>` and `<complex.h>` and defines several type-generic macros.

2 Of the `<math.h>` and `<complex.h>` functions without an `f` (`float`) or `l` (`long double`) suffix, several have one or more parameters whose corresponding real type is `double`. For each such function, except `modf`, there is a corresponding *type-generic macro*.[313] The parameters whose corresponding real type is `double` in the function synopsis are *generic parameters*. Use of the macro invokes a function whose corresponding real type and type domain are determined by the arguments for the generic parameters.[314]

3 Use of the macro invokes a function whose generic parameters have the corresponding real type determined as follows:

— First, if any argument for generic parameters has type `long double`, the type determined is `long double`.

— Otherwise, if any argument for generic parameters has type `double` or is of integer type, the type determined is `double`.

— Otherwise, the type determined is `float`.

4 For each unsuffixed function in `<math.h>` for which there is a function in `<complex.h>` with the same name except for a `c` prefix, the corresponding type-generic macro (for both functions) has the same name as the function in `<math.h>`. The corresponding type-generic macro for `fabs` and `cabs` is `fabs`.

313) Like other function-like macros in Standard libraries, each type-generic macro can be suppressed to make available the corresponding ordinary function.

314) If the type of the argument is not compatible with the type of the parameter for the selected function, the behavior is undefined.

<math.h> function	**<complex.h>** function	type-generic macro
acos	**cacos**	**acos**
asin	**casin**	**asin**
atan	**catan**	**atan**
acosh	**cacosh**	**acosh**
asinh	**casinh**	**asinh**
atanh	**catanh**	**atanh**
cos	**ccos**	**cos**
sin	**csin**	**sin**
tan	**ctan**	**tan**
cosh	**ccosh**	**cosh**
sinh	**csinh**	**sinh**
tanh	**ctanh**	**tanh**
exp	**cexp**	**exp**
log	**clog**	**log**
pow	**cpow**	**pow**
sqrt	**csqrt**	**sqrt**
fabs	**cabs**	**fabs**

If at least one argument for a generic parameter is complex, then use of the macro invokes a complex function; otherwise, use of the macro invokes a real function.

5 For each unsuffixed function in **<math.h>** without a **c**-prefixed counterpart in **<complex.h>** (except **modf**), the corresponding type-generic macro has the same name as the function. These type-generic macros are:

atan2	**fma**	**llround**	**remainder**
cbrt	**fmax**	**log10**	**remquo**
ceil	**fmin**	**log1p**	**rint**
copysign	**fmod**	**log2**	**round**
erf	**frexp**	**logb**	**scalbn**
erfc	**hypot**	**lrint**	**scalbln**
exp2	**ilogb**	**lround**	**tgamma**
expm1	**ldexp**	**nearbyint**	**trunc**
fdim	**lgamma**	**nextafter**	
floor	**llrint**	**nexttoward**	

If all arguments for generic parameters are real, then use of the macro invokes a real function; otherwise, use of the macro results in undefined behavior.

6 For each unsuffixed function in **<complex.h>** that is not a **c**-prefixed counterpart to a function in **<math.h>**, the corresponding type-generic macro has the same name as the function. These type-generic macros are:

carg	**conj**	**creal**
cimag	**cproj**	

Use of the macro with any real or complex argument invokes a complex function.

7 EXAMPLE With the declarations

```
#include <tgmath.h>
int n;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
```

functions invoked by use of type-generic macros are shown in the following table:

macro use	invokes
exp(n)	**exp(n)**, the function
acosh(f)	**acoshf(f)**
sin(d)	**sin(d)**, the function
atan(ld)	**atanl(ld)**
log(fc)	**clogf(fc)**
sqrt(dc)	**csqrt(dc)**
pow(ldc, f)	**cpowl(ldc, f)**
remainder(n, n)	**remainder(n, n)**, the function
nextafter(d, f)	**nextafter(d, f)**, the function
nexttoward(f, ld)	**nexttowardf(f, ld)**
copysign(n, ld)	**copysignl(n, ld)**
ceil(fc)	undefined behavior
rint(dc)	undefined behavior
fmax(ldc, ld)	undefined behavior
carg(n)	**carg(n)**, the function
cproj(f)	**cprojf(f)**
creal(d)	**creal(d)**, the function
cimag(ld)	**cimagl(ld)**
fabs(fc)	**cabsf(fc)**
carg(dc)	**carg(dc)**, the function
cproj(ldc)	**cprojl(ldc)**

7.26 Threads **<threads.h>**

7.26.1 Introduction

1 The header **<threads.h>** includes the header **<time.h>**, defines macros, and declares types, enumeration constants, and functions that support multiple threads of execution.[315)]

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

3 The macros are

```
thread_local
```

which expands to **_Thread_local**;

```
ONCE_FLAG_INIT
```

which expands to a value that can be used to initialize an object of type **once_flag**; and

```
TSS_DTOR_ITERATIONS
```

which expands to an integer constant expression representing the maximum number of times that destructors will be called when a thread terminates.

4 The types are

```
cnd_t
```

which is a complete object type that holds an identifier for a condition variable;

```
thrd_t
```

which is a complete object type that holds an identifier for a thread;

```
tss_t
```

which is a complete object type that holds an identifier for a thread-specific storage pointer;

```
mtx_t
```

which is a complete object type that holds an identifier for a mutex;

```
tss_dtor_t
```

which is the function pointer type **void (*)(void*)**, used for a destructor for a thread-specific storage pointer;

---

315) See ‘‘future library directions’’ (7.31.15).

thrd_start_t

which is the function pointer type **int (*)(void*)** that is passed to **thrd_create** to create a new thread; and

once_flag

which is a complete object type that holds a flag for use by **call_once**.

5 The enumeration constants are

mtx_plain

which is passed to **mtx_init** to create a mutex object that supports neither timeout nor test and return;

mtx_recursive

which is passed to **mtx_init** to create a mutex object that supports recursive locking;

mtx_timed

which is passed to **mtx_init** to create a mutex object that supports timeout;

thrd_timedout

which is returned by a timed wait function to indicate that the time specified in the call was reached without acquiring the requested resource;

thrd_success

which is returned by a function to indicate that the requested operation succeeded;

thrd_busy

which is returned by a function to indicate that the requested operation failed because a resource requested by a test and return function is already in use;

thrd_error

which is returned by a function to indicate that the requested operation failed; and

thrd_nomem

which is returned by a function to indicate that the requested operation failed because it was unable to allocate memory.

Forward references: date and time (7.27).

7.26.2 Initialization functions

7.26.2.1 The `call_once` function

Synopsis

1
```
#include <threads.h>
void call_once(once_flag *flag, void (*func)(void));
```

Description

2 The **call_once** function uses the **once_flag** pointed to by **flag** to ensure that **func** is called exactly once, the first time the **call_once** function is called with that value of **flag**. Completion of an effective call to the **call_once** function synchronizes with all subsequent calls to the **call_once** function with the same value of **flag**.

Returns

3 The **call_once** function returns no value.

7.26.3 Condition variable functions

7.26.3.1 The `cnd_broadcast` function

Synopsis

1
```
#include <threads.h>
int cnd_broadcast(cnd_t *cond);
```

Description

2 The **cnd_broadcast** function unblocks all of the threads that are blocked on the condition variable pointed to by **cond** at the time of the call. If no threads are blocked on the condition variable pointed to by **cond** at the time of the call, the function does nothing.

Returns

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

7.26.3.2 The `cnd_destroy` function

Synopsis

1
```
#include <threads.h>
void cnd_destroy(cnd_t *cond);
```

Description

2 The **cnd_destroy** function releases all resources used by the condition variable pointed to by **cond**. The **cnd_destroy** function requires that no threads be blocked waiting for the condition variable pointed to by **cond**.

Returns

3 The `cnd_destroy` function returns no value.

7.26.3.3 The `cnd_init` function

Synopsis

1
```
#include <threads.h>
int cnd_init(cnd_t *cond);
```

Description

2 The `cnd_init` function creates a condition variable. If it succeeds it sets the variable pointed to by `cond` to a value that uniquely identifies the newly created condition variable. A thread that calls `cnd_wait` on a newly created condition variable will block.

Returns

3 The `cnd_init` function returns `thrd_success` on success, or `thrd_nomem` if no memory could be allocated for the newly created condition, or `thrd_error` if the request could not be honored.

7.26.3.4 The `cnd_signal` function

Synopsis

1
```
#include <threads.h>
int cnd_signal(cnd_t *cond);
```

Description

2 The `cnd_signal` function unblocks one of the threads that are blocked on the condition variable pointed to by `cond` at the time of the call. If no threads are blocked on the condition variable at the time of the call, the function does nothing and return success.

Returns

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

7.26.3.5 The `cnd_timedwait` function

Synopsis

1
```
#include <threads.h>
int cnd_timedwait(cnd_t *restrict cond,
     mtx_t *restrict mtx,
     const struct timespec *restrict ts);
```

Description

2 The **`cnd_timedwait`** function atomically unlocks the mutex pointed to by **`mtx`** and endeavors to block until the condition variable pointed to by **`cond`** is signaled by a call to **`cnd_signal`** or to **`cnd_broadcast`**, or until after the **`TIME_UTC`**-based calendar time pointed to by **`ts`**. When the calling thread becomes unblocked it locks the variable pointed to by **`mtx`** before it returns. The **`cnd_timedwait`** function requires that the mutex pointed to by **`mtx`** be locked by the calling thread.

Returns

3 The **`cnd_timedwait`** function returns **`thrd_success`** upon success, or **`thrd_timedout`** if the time specified in the call was reached without acquiring the requested resource, or **`thrd_error`** if the request could not be honored.

7.26.3.6 The `cnd_wait` function

Synopsis

1
```
#include <threads.h>
int cnd_wait(cnd_t *cond, mtx_t *mtx);
```

Description

2 The **`cnd_wait`** function atomically unlocks the mutex pointed to by **`mtx`** and endeavors to block until the condition variable pointed to by **`cond`** is signaled by a call to **`cnd_signal`** or to **`cnd_broadcast`**. When the calling thread becomes unblocked it locks the mutex pointed to by **`mtx`** before it returns. The **`cnd_wait`** function requires that the mutex pointed to by **`mtx`** be locked by the calling thread.

Returns

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

7.26.4 Mutex functions

7.26.4.1 The `mtx_destroy` function

Synopsis

1
```
#include <threads.h>
void mtx_destroy(mtx_t *mtx);
```

Description

2 The **`mtx_destroy`** function releases any resources used by the mutex pointed to by **`mtx`**. No threads can be blocked waiting for the mutex pointed to by **`mtx`**.

Returns

3 The **`mtx_destroy`** function returns no value.

7.26.4.2 The `mtx_init` function

Synopsis

1
```
#include <threads.h>
int mtx_init(mtx_t *mtx, int type);
```

Description

2 The `mtx_init` function creates a mutex object with properties indicated by `type`, which must have one of the six values:

`mtx_plain` for a simple non-recursive mutex,
`mtx_timed` for a non-recursive mutex that supports timeout,
`mtx_plain | mtx_recursive` for a simple recursive mutex, or
`mtx_timed | mtx_recursive` for a recursive mutex that supports timeout.

3 If the `mtx_init` function succeeds, it sets the mutex pointed to by `mtx` to a value that uniquely identifies the newly created mutex.

Returns

4 The `mtx_init` function returns `thrd_success` on success, or `thrd_error` if the request could not be honored.

7.26.4.3 The `mtx_lock` function

Synopsis

1
```
#include <threads.h>
int mtx_lock(mtx_t *mtx);
```

Description

2 The `mtx_lock` function blocks until it locks the mutex pointed to by `mtx`. If the mutex is non-recursive, it shall not be locked by the calling thread. Prior calls to `mtx_unlock` on the same mutex shall synchronize with this operation.

Returns

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

7.26.4.4 The `mtx_timedlock` function

Synopsis

1
```
#include <threads.h>
int mtx_timedlock(mtx_t *restrict mtx,
     const struct timespec *restrict ts);
```

Description

2 The **`mtx_timedlock`** function endeavors to block until it locks the mutex pointed to by **`mtx`** or until after the **`TIME_UTC`**-based calendar time pointed to by **`ts`**. The specified mutex shall support timeout. If the operation succeeds, prior calls to **`mtx_unlock`** on the same mutex shall synchronize with this operation.

Returns

3 The **`mtx_timedlock`** function returns **`thrd_success`** on success, or **`thrd_timedout`** if the time specified was reached without acquiring the requested resource, or **`thrd_error`** if the request could not be honored.

7.26.4.5 The `mtx_trylock` function

Synopsis

1
```
#include <threads.h>
int mtx_trylock(mtx_t *mtx);
```

Description

2 The **`mtx_trylock`** function endeavors to lock the mutex pointed to by **`mtx`**. If the mutex is already locked, the function returns without blocking. If the operation succeeds, prior calls to **`mtx_unlock`** on the same mutex shall synchronize with this operation.

Returns

3 The **`mtx_trylock`** function returns **`thrd_success`** on success, or **`thrd_busy`** if the resource requested is already in use, or **`thrd_error`** if the request could not be honored.

7.26.4.6 The `mtx_unlock` function

Synopsis

1
```
#include <threads.h>
int mtx_unlock(mtx_t *mtx);
```

Description

2 The **`mtx_unlock`** function unlocks the mutex pointed to by **`mtx`**. The mutex pointed to by **`mtx`** shall be locked by the calling thread.

Returns

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

7.26.5 Thread functions

7.26.5.1 The `thrd_create` function

Synopsis

1
```
#include <threads.h>
int thrd_create(thrd_t *thr, thrd_start_t func,
     void *arg);
```

Description

2 The `thrd_create` function creates a new thread executing `func(arg)`. If the `thrd_create` function succeeds, it sets the object pointed to by `thr` to the identifier of the newly created thread. (A thread's identifier may be reused for a different thread once the original thread has exited and either been detached or joined to another thread.) The completion of the `thrd_create` function synchronizes with the beginning of the execution of the new thread.

Returns

3 The `thrd_create` function returns `thrd_success` on success, or `thrd_nomem` if no memory could be allocated for the thread requested, or `thrd_error` if the request could not be honored.

7.26.5.2 The `thrd_current` function

Synopsis

1
```
#include <threads.h>
thrd_t thrd_current(void);
```

Description

2 The `thrd_current` function identifies the thread that called it.

Returns

3 The `thrd_current` function returns the identifier of the thread that called it.

7.26.5.3 The `thrd_detach` function

Synopsis

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

Description

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

Returns

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

7.26.5.4 The thrd_equal function

Synopsis

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

Description

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

Returns

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

7.26.5.5 The thrd_exit function

Synopsis

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

Description

2 The **thrd_exit** function terminates execution of the calling thread and sets its result code to **res**.

3 The program shall terminate normally after the last thread has been terminated. The behavior shall be as if the program called the **exit** function with the status **EXIT_SUCCESS** at thread termination time.

Returns

4 The **thrd_exit** function returns no value.

7.26.5.6 The thrd_join function

Synopsis

1
```
#include <threads.h>
int thrd_join(thrd_t thr, int *res);
```

Description

2 The **thrd_join** function joins the thread identified by **thr** with the current thread by blocking until the other thread has terminated. If the parameter **res** is not a null pointer, it stores the thread's result code in the integer pointed to by **res**. The termination of the

other thread synchronizes with the completion of the **thrd_join** function. The thread identified by **thr** shall not have been previously detached or joined with another thread.

Returns

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

7.26.5.7 The thrd_sleep function

Synopsis

1
```
#include <threads.h>
int thrd_sleep(const struct timespec *duration,
     struct timespec *remaining);
```

Description

2 The **thrd_sleep** function suspends execution of the calling thread until either the interval specified by **duration** has elapsed or a signal which is not being ignored is received. If interrupted by a signal and the **remaining** argument is not null, the amount of time remaining (the requested interval minus the time actually slept) is stored in the interval it points to. The **duration** and **remaining** arguments may point to the same object.

3 The suspension time may be longer than requested because the interval is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time shall not be less than that specified, as measured by the system clock **TIME_UTC**.

Returns

4 The **thrd_sleep** function returns zero if the requested time has elapsed, -1 if it has been interrupted by a signal, or a negative value if it fails.

7.26.5.8 The thrd_yield function

Synopsis

1
```
#include <threads.h>
void thrd_yield(void);
```

Description

2 The **thrd_yield** function endeavors to permit other threads to run, even if the current thread would ordinarily continue to run.

Returns

3 The **thrd_yield** function returns no value.

7.26.6 Thread-specific storage functions

7.26.6.1 The `tss_create` function

Synopsis

1
```
#include <threads.h>
int tss_create(tss_t *key, tss_dtor_t dtor);
```

Description

2 The **`tss_create`** function creates a thread-specific storage pointer with destructor **`dtor`**, which may be null.

Returns

3 If the **`tss_create`** function is successful, it sets the thread-specific storage pointed to by **`key`** to a value that uniquely identifies the newly created pointer and returns **`thrd_success`**; otherwise, **`thrd_error`** is returned and the thread-specific storage pointed to by **`key`** is set to an undefined value.

7.26.6.2 The `tss_delete` function

Synopsis

1
```
#include <threads.h>
void tss_delete(tss_t key);
```

Description

2 The **`tss_delete`** function releases any resources used by the thread-specific storage identified by **`key`**.

Returns

3 The **`tss_delete`** function returns no value.

7.26.6.3 The `tss_get` function

Synopsis

1
```
#include <threads.h>
void *tss_get(tss_t key);
```

Description

2 The **`tss_get`** function returns the value for the current thread held in the thread-specific storage identified by **`key`**.

Returns

3 The **`tss_get`** function returns the value for the current thread if successful, or zero if unsuccessful.

7.26.6.4 The `tss_set` function

Synopsis

1
```
#include <threads.h>
int tss_set(tss_t key, void *val);
```

Description

2 The **`tss_set`** function sets the value for the current thread held in the thread-specific storage identified by **`key`** to **`val`**.

Returns

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

7.27 Date and time <time.h>

7.27.1 Components of time

1 The header **<time.h>** defines two macros, and declares several types and functions for manipulating time. Many functions deal with a *calendar time* that represents the current date (according to the Gregorian calendar) and time. Some functions deal with *local time*, which is the calendar time expressed for some specific time zone, and with *Daylight Saving Time*, which is a temporary change in the algorithm for determining local time. The local time zone and Daylight Saving Time are implementation-defined.

2 The macros defined are **NULL** (described in 7.19); *

CLOCKS_PER_SEC

which expands to an expression with type **clock_t** (described below) that is the number per second of the value returned by the **clock** function; and

TIME_UTC

which expands to an integer constant greater than 0 that designates the UTC time base.[316)]

3 The types declared are **size_t** (described in 7.19);

clock_t

and

time_t

which are real types capable of representing times;

struct timespec

which holds an interval specified in seconds and nanoseconds (which may represent a calendar time based on a particular epoch); and

struct tm

which holds the components of a calendar time, called the *broken-down time*.

4 The range and precision of times representable in **clock_t** and **time_t** are implementation-defined. The **timespec** structure shall contain at least the following members, in any order.[317)]

---

316) Implementations may define additional time bases, but are only required to support a real time clock based on UTC.

317) The **tv_sec** member is a linear count of seconds and may not have the normal semantics of a **time_t**. The semantics of the members and their normal ranges are expressed in the comments.

```
time_t tv_sec;   // whole seconds — ≥ 0
long   tv_nsec;  // nanoseconds — [0, 999999999]
```

The **tm** structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.[318)]

```
int tm_sec;    // seconds after the minute — [0, 60]
int tm_min;    // minutes after the hour — [0, 59]
int tm_hour;   // hours since midnight — [0, 23]
int tm_mday;   // day of the month — [1, 31]
int tm_mon;    // months since January — [0, 11]
int tm_year;   // years since 1900
int tm_wday;   // days since Sunday — [0, 6]
int tm_yday;   // days since January 1 — [0, 365]
int tm_isdst;  // Daylight Saving Time flag
```

The value of **tm_isdst** is positive if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and negative if the information is not available.

7.27.2 Time manipulation functions

7.27.2.1 The clock function

Synopsis

1
```
#include <time.h>
clock_t clock(void);
```

Description

2 The **clock** function determines the processor time used.

Returns

3 The **clock** function returns the implementation's best approximation to the processor time used by the program since the beginning of an implementation-defined era related only to the program invocation. To determine the time in seconds, the value returned by the **clock** function should be divided by the value of the macro **CLOCKS_PER_SEC**. If the processor time used is not available or its value cannot be represented, the function returns the value **(clock_t)(-1)**.[319)]

318) The range [0, 60] for **tm_sec** allows for a positive leap second.

319) In order to measure the time spent in a program, the **clock** function should be called at the start of the program and its return value subtracted from the value returned by subsequent calls.

7.27.2.2 The `difftime` function

Synopsis

1
```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

Description

2 The **`difftime`** function computes the difference between two calendar times: **`time1 - time0`**.

Returns

3 The **`difftime`** function returns the difference expressed in seconds as a **`double`**.

7.27.2.3 The `mktime` function

Synopsis

1
```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

Description

2 The **`mktime`** function converts the broken-down time, expressed as local time, in the structure pointed to by **`timeptr`** into a calendar time value with the same encoding as that of the values returned by the **`time`** function. The original values of the **`tm_wday`** and **`tm_yday`** components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above.[320] On successful completion, the values of the **`tm_wday`** and **`tm_yday`** components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of **`tm_mday`** is not set until **`tm_mon`** and **`tm_year`** are determined.

Returns

3 The **`mktime`** function returns the specified calendar time encoded as a value of type **`time_t`**. If the calendar time cannot be represented, the function returns the value **`(time_t)(-1)`**.

4 EXAMPLE What day of the week is July 4, 2001?

320) Thus, a positive or zero value for **`tm_isdst`** causes the **`mktime`** function to presume initially that Daylight Saving Time, respectively, is or is not in effect for the specified time. A negative value causes it to attempt to determine whether Daylight Saving Time is in effect for the specified time.

```
#include <stdio.h>
#include <time.h>
static const char *const wday[] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday", "-unknown-"
};
struct tm time_str;
/* ... */

time_str.tm_year   = 2001 - 1900;
time_str.tm_mon    = 7 - 1;
time_str.tm_mday   = 4;
time_str.tm_hour   = 0;
time_str.tm_min    = 0;
time_str.tm_sec    = 1;
time_str.tm_isdst  = -1;
if (mktime(&time_str) == (time_t)(-1))
      time_str.tm_wday = 7;
printf("%s\n", wday[time_str.tm_wday]);
```

7.27.2.4 The `time` function

Synopsis

1
```
#include <time.h>
time_t time(time_t *timer);
```

Description

2 The **`time`** function determines the current calendar time. The encoding of the value is unspecified.

Returns

3 The **`time`** function returns the implementation's best approximation to the current calendar time. The value **`(time_t)(-1)`** is returned if the calendar time is not available. If **`timer`** is not a null pointer, the return value is also assigned to the object it points to.

7.27.2.5 The `timespec_get` function

Synopsis

1
```
#include <time.h>
int timespec_get(struct timespec *ts, int base);
```

Description

2 The **`timespec_get`** function sets the interval pointed to by **`ts`** to hold the current calendar time based on the specified time base.

3 If **`base`** is **`TIME_UTC`**, the **`tv_sec`** member is set to the number of seconds since an implementation defined *epoch*, truncated to a whole value and the **`tv_nsec`** member is set to the integral number of nanoseconds, rounded to the resolution of the system

clock.[321)]

Returns

4 If the **timespec_get** function is successful it returns the nonzero value **base**; otherwise, it returns zero.

7.27.3 Time conversion functions

1 Except for the **strftime** function, these functions each return a pointer to one of two types of static objects: a broken-down time structure or an array of **char**. Execution of any of the functions that return a pointer to one of these object types may overwrite the information in any object of the same type pointed to by the value returned from any previous call to any of them and the functions are not required to avoid data races with each other.[322)] The implementation shall behave as if no other library functions call these functions.

7.27.3.1 The asctime function

Synopsis

1
```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

Description

2 The **asctime** function converts the broken-down time in the structure pointed to by **timeptr** into a string in the form

```
Sun Sep 16 01:03:52 1973\n\0
```

using the equivalent of the following algorithm.

```
char *asctime(const struct tm *timeptr)
{
     static const char wday_name[7][3] = {
          "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
     };
     static const char mon_name[12][3] = {
          "Jan", "Feb", "Mar", "Apr", "May", "Jun",
          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
     };
     static char resul875l.
     n5har resul875l.
```

```
        sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
             wday_name[timeptr->tm_wday],
             mon_name[timeptr->tm_mon],
             timeptr->tm_mday, timeptr->tm_hour,
             timeptr->tm_min, timeptr->tm_sec,
             1900 + timeptr->tm_year);
        return result;
}
```

3 If any of the members of the broken-down time contain values that are outside their normal ranges,[323)] the behavior of the **asctime** function is undefined. Likewise, if the calculated year exceeds four digits or is less than the year 1000, the behavior is undefined.

Returns

4 The **asctime** function returns a pointer to the string.

7.27.3.2 The ctime function

Synopsis

1
```
#include <time.h>
char *ctime(const time_t *timer);
```

Description

2 The **ctime** function converts the calendar time pointed to by **timer** to local time in the form of a string. It is equivalent to

```
asctime(localtime(timer))
```

Returns

3 The **ctime** function returns the pointer returned by the **asctime** function with that broken-down time as argument.

Forward references: the **localtime** function (7.27.3.4).

7.27.3.3 The gmtime function

Synopsis

1
```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

323) See 7.27.1.

Description

2 The **`gmtime`** function converts the calendar time pointed to by **`timer`** into a broken-down time, expressed as UTC.

Returns

3 The **`gmtime`** function returns a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to UTC.

7.27.3.4 The `localtime` function

Synopsis

1
```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

Description

2 The **`localtime`** function converts the calendar time pointed to by **`timer`** into a broken-down time, expressed as local time.

Returns

3 The **`localtime`** function returns a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to local time.

7.27.3.5 The `strftime` function

Synopsis

1
```
#include <time.h>
size_t strftime(char * restrict s,
     size_t maxsize,
     const char * restrict format,
     const struct tm * restrict timeptr);
```

Description

2 The **`strftime`** function places characters into the array pointed to by **`s`** as controlled by the string pointed to by **`format`**. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The **`format`** string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a **`%`** character, possibly followed by an **`E`** or **`O`** modifier character (described below), followed by a character that determines the behavior of the conversion specifier. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than **`maxsize`** characters are placed into the array.

3 Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined using the **`LC_TIME`** category

of the current locale and by the values of zero or more members of the broken-down time structure pointed to by **timeptr**, as specified in brackets in the description. If any of the specified values is outside the normal range, the characters stored are unspecified.

%a is replaced by the locale's abbreviated weekday name. [**tm_wday**]
%A is replaced by the locale's full weekday name. [**tm_wday**]
%b is replaced by the locale's abbreviated month name. [**tm_mon**]
%B is replaced by the locale's full month name. [**tm_mon**]
%c is replaced by the locale's appropriate date and time representation. [all specified in 7.27.1]
%C is replaced by the year divided by 100 and truncated to an integer, as a decimal number (**00**−**99**). [**tm_year**]
%d is replaced by the day of the month as a decimal number (**01**−**31**). [**tm_mday**]
%D is equivalent to "**%m/%d/%y**". [**tm_mon**, **tm_mday**, **tm_year**]
%e is replaced by the day of the month as a decimal number (**1**−**31**); a single digit is preceded by a space. [**tm_mday**]
%F is equivalent to "**%Y-%m-%d**" (the ISO 8601 date format). [**tm_year**, **tm_mon**, **tm_mday**]
%g is replaced by the last 2 digits of the week-based year (see below) as a decimal number (**00**−**99**). [**tm_year**, **tm_wday**, **tm_yday**]
%G is replaced by the week-based year (see below) as a decimal number (e.g., 1997). [**tm_year**, **tm_wday**, **tm_yday**]
%h is equivalent to "**%b**". [**tm_mon**]
%H is replaced by the hour (24-hour clock) as a decimal number (**00**−**23**). [**tm_hour**]
%I is replaced by the hour (12-hour clock) as a decimal number (**01**−**12**). [**tm_hour**]
%j is replaced by the day of the year as a decimal number (**001**−**366**). [**tm_yday**]
%m is replaced by the month as a decimal number (**01**−**12**). [**tm_mon**]
%M is replaced by the minute as a decimal number (**00**−**59**). [**tm_min**]
%n is replaced by a new-line character.
%p is replaced by the locale's equivalent of the AM/PM designations associated with a 12-hour clock. [**tm_hour**]
%r is replaced by the locale's 12-hour clock time. [**tm_hour**, **tm_min**, **tm_sec**]
%R is equivalent to "**%H:%M**". [**tm_hour**, **tm_min**]
%S is replaced by the second as a decimal number (**00**−**60**). [**tm_sec**]
%t is replaced by a horizontal-tab character.
%T is equivalent to "**%H:%M:%S**" (the ISO 8601 time format). [**tm_hour**, **tm_min**, **tm_sec**]
%u is replaced by the ISO 8601 weekday as a decimal number (**1**−**7**), where Monday is 1. [**tm_wday**]
%U is replaced by the week number of the year (the first Sunday as the first day of week 1) as a decimal number (**00**−**53**). [**tm_year**, **tm_wday**, **tm_yday**]
%V is replaced by the ISO 8601 week number (see below) as a decimal number

(**01**−**53**). [**tm_year**, **tm_wday**, **tm_yday**]

%w is replaced by the weekday as a decimal number (**0**−**6**), where Sunday is 0. [**tm_wday**]

%W is replaced by the week number of the year (the first Monday as the first day of week 1) as a decimal number (**00**−**53**). [**tm_year**, **tm_wday**, **tm_yday**]

%x is replaced by the locale's appropriate date representation. [all specified in 7.27.1]

%X is replaced by the locale's appropriate time representation. [all specified in 7.27.1]

%y is replaced by the last 2 digits of the year as a decimal number (**00**−**99**). [**tm_year**]

%Y is replaced by the year as a decimal number (e.g., **1997**). [**tm_year**]

%z is replaced by the offset from UTC in the ISO 8601 format ''**-0430**'' (meaning 4 hours 30 minutes behind UTC, west of Greenwich), or by no characters if no time zone is determinable. [**tm_isdst**]

%Z is replaced by the locale's time zone name or abbreviation, or by no characters if no time zone is determinable. [**tm_isdst**]

%% is replaced by **%**.

4 Some conversion specifiers can be modified by the inclusion of an **E** or **O** modifier character to indicate an alternative format or specification. If the alternative format or specification does not exist for the current locale, the modifier is ignored.

%Ec is replaced by the locale's alternative date and time representation.

%EC is replaced by the name of the base year (period) in the locale's alternative representation.

%Ex is replaced by the locale's alternative date representation.

%EX is replaced by the locale's alternative time representation.

%Ey is replaced by the offset from **%EC** (year only) in the locale's alternative representation.

%EY is replaced by the locale's full alternative year representation.

%Od is replaced by the day of the month, using the locale's alternative numeric symbols (filled as needed with leading zeros, or with leading spaces if there is no alternative symbol for zero).

%Oe is replaced by the day of the month, using the locale's alternative numeric symbols (filled as needed with leading spaces).

%OH is replaced by the hour (24-hour clock), using the locale's alternative numeric symbols.

%OI is replaced by the hour (12-hour clock), using the locale's alternative numeric symbols.

%Om is replaced by the month, using the locale's alternative numeric symbols.

%OM is replaced by the minutes, using the locale's alternative numeric symbols.

%OS is replaced by the seconds, using the locale's alternative numeric symbols.

%Ou is replaced by the ISO 8601 weekday as a number in the locale's alternative representation, where Monday is 1.

%OU is replaced by the week number, using the locale's alternative numeric symbols.

%OV is replaced by the ISO 8601 week number, using the locale's alternative numeric symbols.

%Ow is replaced by the weekday as a number, using the locale's alternative numeric symbols.

%OW is replaced by the week number of the year, using the locale's alternative numeric symbols.

%Oy is replaced by the last 2 digits of the year, using the locale's alternative numeric symbols.

5 **%g**, **%G**, and **%V** give values according to the ISO 8601 week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, **%G** is replaced by **1998** and **%V** is replaced by **53**. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, **%G** is replaced by **1998** and **%V** is replaced by **01**.

6 If a conversion specifier is not one of the above, the behavior is undefined.

7 In the **"C"** locale, the **E** and **O** modifiers are ignored and the replacement strings for the following specifiers are:

%a the first three characters of **%A**.

%A one of "**Sunday**", "**Monday**", ... , "**Saturday**".

%b the first three characters of **%B**.

%B one of "**January**", "**February**", ... , "**December**".

%c equivalent to "**%a %b %e %T %Y**".

%p one of "**AM**" or "**PM**".

%r equivalent to "**%I:%M:%S %p**".

%x equivalent to "**%m/%d/%y**".

%X equivalent to **%T**.

%Z implementation-defined.

Returns

8 If the total number of resulting characters including the terminating null character is not more than **maxsize**, the **strftime** function returns the number of characters placed into the array pointed to by **s** not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

7.28 Unicode utilities <uchar.h>

1 The header **<uchar.h>** declares types and functions for manipulating Unicode characters.

2 The types declared are **mbstate_t** (described in 7.30.1) and **size_t** (described in 7.19);

char16_t

which is an unsigned integer type used for 16-bit characters and is the same type as **uint_least16_t** (described in 7.20.1.2); and

char32_t

which is an unsigned integer type used for 32-bit characters and is the same type as **uint_least32_t** (also described in 7.20.1.2).

7.28.1 Restartable multibyte/wide character conversion functions

1 These functions have a parameter, **ps**, of type pointer to **mbstate_t** that points to an object that can completely describe the current conversion state of the associated multibyte character sequence, which the functions alter as necessary. If **ps** is a null pointer, each function uses its own internal **mbstate_t** object instead, which is initialized at program startup to the initial conversion state; the functions are not required to avoid data races with other calls to the same function in this case. The implementation behaves as if no library function calls these functions with a null pointer for **ps**.

7.28.1.1 The mbrtoc16 function

Synopsis

1
```
#include <uchar.h>
size_t mbrtoc16(char16_t * restrict pc16,
     const char * restrict s, size_t n,
     mbstate_t * restrict ps);
```

Description

2 If **s** is a null pointer, the **mbrtoc16** function is equivalent to the call:

```
mbrtoc16(NULL, "", 1, ps)
```

In this case, the values of the parameters **pc16** and **n** are ignored.

3 If **s** is not a null pointer, the **mbrtoc16** function inspects at most **n** bytes beginning with the byte pointed to by **s** to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the values of the corresponding wide characters and then, if **pc16** is not a null pointer, stores the value of the first (or only) such character in the object pointed to by **pc16**. Subsequent calls will

store successive wide characters without consuming any additional input until all the characters have been stored. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

4 The `mbrtoc16` function returns the first of the following that applies (given the current conversion state):

0 if the next `n` or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 *and* `n` *inclusive* if the next `n` or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

`(size_t)(-3)` if the next character resulting from a previous call has been stored (no bytes from the input have been consumed by this call).

`(size_t)(-2)` if the next `n` bytes contribute to an incomplete (but potentially valid) multibyte character, and all `n` bytes have been processed (no value is stored).[324)]

`(size_t)(-1)` if an encoding error occurs, in which case the next `n` or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro `EILSEQ` is stored in `errno`, and the conversion state is unspecified.

7.28.1.2 The `c16rtomb` function

Synopsis

1
```
#include <uchar.h>
size_t c16rtomb(char * restrict s, char16_t c16,
     mbstate_t * restrict ps);
```

Description

2 If `s` is a null pointer, the `c16rtomb` function is equivalent to the call

```
c16rtomb(buf, L'\0', ps)
```

where `buf` is an internal buffer.

3 If `s` is not a null pointer, the `c16rtomb` function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by `c16` (including any shift sequences), and stores the multibyte character representation in the

324) When `n` has at least the value of the `MB_CUR_MAX` macro, this case can only occur if `s` points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

array whose first element is pointed to by **s**. At most **MB_CUR_MAX** bytes are stored. If **c16** is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

4 The **c16rtomb** function returns the number of bytes stored in the array object (including any shift sequences). When **c16** is not a valid wide character, an encoding error occurs: the function stores the value of the macro **EILSEQ** in **errno** and returns **(size_t)(-1)**; the conversion state is unspecified.

7.28.1.3 The mbrtoc32 function

Synopsis

1
```
#include <uchar.h>
size_t mbrtoc32(char32_t * restrict pc32,
     const char * restrict s, size_t n,
     mbstate_t * restrict ps);
```

Description

2 If **s** is a null pointer, the **mbrtoc32** function is equivalent to the call:

```
mbrtoc32(NULL, "", 1, ps)
```

In this case, the values of the parameters **pc32** and **n** are ignored.

3 If **s** is not a null pointer, the **mbrtoc32** function inspects at most **n** bytes beginning with the byte pointed to by **s** to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the values of the corresponding wide characters and then, if **pc32** is not a null pointer, stores the value of the first (or only) such character in the object pointed to by **pc32**. Subsequent calls will store successive wide characters without consuming any additional input until all the characters have been stored. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

4 The **mbrtoc32** function returns the first of the following that applies (given the current conversion state):

0 if the next **n** or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 *and* **n** *inclusive* if the next **n** or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

(size_t)(-3) if the next character resulting from a previous call has been stored (no bytes from the input have been consumed by this call).

(size_t)(-2) if the next **n** bytes contribute to an incomplete (but potentially valid) multibyte character, and all **n** bytes have been processed (no value is stored).[325)]

(size_t)(-1) if an encoding error occurs, in which case the next **n** or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro **EILSEQ** is stored in **errno**, and the conversion state is unspecified.

7.28.1.4 The c32rtomb function

Synopsis

1
```
#include <uchar.h>
size_t c32rtomb(char * restrict s, char32_t c32,
     mbstate_t * restrict ps);
```

Description

2 If **s** is a null pointer, the **c32rtomb** function is equivalent to the call

```
c32rtomb(buf, L'\0', ps)
```

where **buf** is an internal buffer.

3 If **s** is not a null pointer, the **c32rtomb** function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by **c32** (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by **s**. At most **MB_CUR_MAX** bytes are stored. If **c32** is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

4 The **c32rtomb** function returns the number of bytes stored in the array object (including any shift sequences). When **c32** is not a valid wide character, an encoding error occurs: the function stores the value of the macro **EILSEQ** in **errno** and returns **(size_t)(-1)**; the conversion state is unspecified.

325) When **n** has at least the value of the **MB_CUR_MAX** macro, this case can only occur if **s** points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

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

7.29.1 Introduction

1 The header **`<wchar.h>`** defines four macros, and declares four data types, one tag, and many functions.[326)]

2 The types declared are **`wchar_t`** and **`size_t`** (both described in 7.19);

```
mbstate_t
```

which is a complete object type other than an array type that can hold the conversion state information necessary to convert between sequences of multibyte characters and wide characters;

```
wint_t
```

which is an integer type unchanged by default argument promotions that can hold any value corresponding to members of the extended character set, as well as at least one value that does not correspond to any member of the extended character set (see **`WEOF`** below);[327)] and

```
struct tm
```

which is declared as an incomplete structure type (the contents are described in 7.27.1).

3 The macros defined are **`NULL`** (described in 7.19); **`WCHAR_MIN`** and **`WCHAR_MAX`** (described in 7.20.3); and

```
WEOF
```

which expands to a constant expression of type **`wint_t`** whose value does not correspond to any member of the extended character set.[328)] It is accepted (and returned) by several functions in this subclause to indicate *end-of-file*, that is, no more input from a stream. It is also used as a wide character value that does not correspond to any member of the extended character set.

4 The functions declared are grouped as follows:

— Functions that perform input and output of wide characters, or multibyte characters, or both;

— Functions that provide wide string numeric conversion;

— Functions that perform general wide string manipulation;

---

326) See ‘‘future library directions’’ (7.31.16).

327) **`wchar_t`** and **`wint_t`** can be the same integer type.

328) The value of the macro **`WEOF`** may differ from that of **`EOF`** and need not be negative.

— Functions for wide string date and time conversion; and

— Functions that provide extended capabilities for conversion between multibyte and wide character sequences.

5 Arguments to the functions in this subclause may point to arrays containing **`wchar_t`** values that do not correspond to members of the extended character set. Such values shall be processed according to the specified semantics, except that it is unspecified whether an encoding error occurs if such a value appears in the format string for a function in 7.29.2 or 7.29.5 and the specified semantics do not require that value to be processed by **`wcrtomb`**.

6 Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the behavior is undefined.

7.29.2 Formatted wide character input/output functions

1 The formatted wide character input/output functions shall behave as if there is a sequence point after the actions associated with each specifier.[329)]

7.29.2.1 The `fwprintf` function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
int fwprintf(FILE * restrict stream,
     const wchar_t * restrict format, ...);
```

Description

2 The **`fwprintf`** function writes output to the stream pointed to by **`stream`**, under control of the wide string pointed to by **`format`** that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **`fwprintf`** function returns when the end of the format string is encountered.

3 The format is composed of zero or more directives: ordinary wide characters (not **`%`**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

329) The **`fwprintf`** functions perform writes to memory for the **`%n`** specifier.

4 Each conversion specification is introduced by the wide character **%**. After the **%**, the following appear in sequence:

— Zero or more *flags* (in any order) that modify the meaning of the conversion specification.

— An optional minimum *field width*. If the converted value has fewer wide characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk ***** (described later) or a nonnegative decimal integer.[330)]

— An optional *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point wide character for **a**, **A**, **e**, **E**, **f**, and **F** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of wide characters to be written for **s** conversions. The precision takes the form of a period (**.**) followed either by an asterisk ***** (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

— An optional *length modifier* that specifies the size of the argument.

— A *conversion specifier* wide character that specifies the type of conversion to be applied.

5 As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an **int** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a **-** flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

6 The flag wide characters and their meanings are:

- The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

+ The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not

330) Note that **0** is taken as a flag, not as the beginning of a field width.

specified.)[331)]

space If the first wide character of a signed conversion is not a sign, or if a signed conversion results in no wide characters, a space is prefixed to the result. If the *space* and **+** flags both appear, the *space* flag is ignored.

The result is converted to an ‘‘alternative form’’. For **o** conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For **x** (or **X**) conversion, a nonzero result has **0x** (or **0X**) prefixed to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result of converting a floating-point number always contains a decimal-point wide character, even if no digits follow it. (Normally, a decimal-point wide character appears in the result of these conversions only if a digit follows it.) For **g** and **G** conversions, trailing zeros are *not* removed from the result. For other conversions, the behavior is undefined.

0 For **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the **0** and **-** flags both appear, the **0** flag is ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversions, if a precision is specified, the **0** flag is ignored. For other conversions, the behavior is undefined.

7 The length modifiers and their meanings are:

hh Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **signed char** or **unsigned char** before printing); or that a following **n** conversion specifier applies to a pointer to a **signed char** argument.

h Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **short int** or **unsigned short int** before printing); or that a following **n** conversion specifier applies to a pointer to a **short int** argument.

l (ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long int** or **unsigned long int** argument; that a following **n** conversion specifier applies to a pointer to a **long int** argument; that a

331) The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

following **c** conversion specifier applies to a **wint_t** argument; that a following **s** conversion specifier applies to a pointer to a **wchar_t** argument; or has no effect on a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier.

ll (ell-ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long long int** or **unsigned long long int** argument; or that a following **n** conversion specifier applies to a pointer to a **long long int** argument.

j Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to an **intmax_t** or **uintmax_t** argument; or that a following **n** conversion specifier applies to a pointer to an **intmax_t** argument.

z Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **size_t** or the corresponding signed integer type argument; or that a following **n** conversion specifier applies to a pointer to a signed integer type corresponding to **size_t** argument.

t Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **ptrdiff_t** or the corresponding unsigned integer type argument; or that a following **n** conversion specifier applies to a pointer to a **ptrdiff_t** argument.

L Specifies that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a **long double** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

8 The conversion specifiers and their meanings are:

d,i The **int** argument is converted to signed decimal in the style *[−]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.

o,u,x,X The **unsigned int** argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**) in the style *dddd*; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.

f,F A **double** argument representing a floating-point number is converted to decimal notation in the style *[−]ddd***.***ddd*, where the number of digits after the decimal-point wide character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point wide character appears. If a decimal-point wide character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

A **double** argument representing an infinity is converted in one of the styles *[-]***inf** or *[-]***infinity** — which style is implementation-defined. A **double** argument representing a NaN is converted in one of the styles *[-]***nan** or *[-]***nan(***n-wchar-sequence***)** — which style, and the meaning of any *n-wchar-sequence*, is implementation-defined. The **F** conversion specifier produces **INF**, **INFINITY**, or **NAN** instead of **inf**, **infinity**, or **nan**, respectively.[332)]

e,E A **double** argument representing a floating-point number is converted in the style *[−]d***.***ddd* **e**±*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point wide character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point wide character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

g,G A **double** argument representing a floating-point number is converted in style **f** or **e** (or in style **F** or **E** in the case of a **G** conversion specifier), depending on the value converted and the precision. Let P equal the precision if nonzero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style **E** would have an exponent of X:

— if $P > X \geq -4$, the conversion is with style **f** (or **F**) and precision $P - (X + 1)$.

— otherwise, the conversion is with style **e** (or **E**) and precision $P - 1$.

Finally, unless the **#** flag is used, any trailing zeros are removed from the

332) When applied to infinite and NaN values, the **-**, **+**, and *space* flag wide characters have their usual meaning; the **#** and **0** flag wide characters have no effect.

fractional portion of the result and the decimal-point wide character is removed if there is no fractional portion remaining.

A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

a,A A **double** argument representing a floating-point number is converted in the style *[*-*]***0x***h***.***hhhh***p**±*d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point wide character[333] and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and **FLT_RADIX** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **FLT_RADIX** is not a power of 2, then the precision is sufficient to distinguish[334] values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the **#** flag is not specified, no decimal-point wide character appears. The letters **abcdef** are used for **a** conversion and the letters **ABCDEF** for **A** conversion. The **A** conversion specifier produces a number with **X** and **P** instead of **x** and **p**. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

c If no **l** length modifier is present, the **int** argument is converted to a wide character as if by calling **btowc** and the resulting wide character is written.

If an **l** length modifier is present, the **wint_t** argument is converted to **wchar_t** and written.

s If no **l** length modifier is present, the argument shall be a pointer to the initial element of a character array containing a multibyte character sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted, and written up to (but not including) the

333) Binary implementations can choose the hexadecimal digit to the left of the decimal-point wide character so that subsequent digits align to nibble (4-bit) boundaries.

334) The precision p is sufficient to distinguish values of the source type if $16^{p-1} > b^n$ where b is **FLT_RADIX** and n is the number of base-b digits in the significand of the source type. A smaller p might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point wide character.

terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the converted array, the converted array shall contain a null wide character.

If an **l** length modifier is present, the argument shall be a pointer to the initial element of an array of **wchar_t** type. Wide characters from the array are written up to (but not including) a terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null wide character.

p The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printing wide characters, in an implementation-defined manner.

n The argument shall be a pointer to signed integer into which is *written* the number of wide characters written to the output stream so far by this call to **fwprintf**. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

% A **%** wide character is written. No argument is converted. The complete conversion specification shall be **%%**.

9 If a conversion specification is invalid, the behavior is undefined.[335)] If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

10 In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

11 For **a** and **A** conversions, if **FLT_RADIX** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

12 For **a** and **A** conversions, if **FLT_RADIX** is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

13 For **e**, **E**, **f**, **F**, **g**, and **G** conversions, if the number of significant decimal digits is at most **DECIMAL_DIG**, then the result should be correctly rounded.[336)] If the number of

335) See ‘‘future library directions’’ (7.31.16).

significant decimal digits is more than **DECIMAL_DIG** but the source value is exactly representable with **DECIMAL_DIG** digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having **DECIMAL_DIG** significant digits; the value of the resultant decimal string D should satisfy $L \le D \le U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

Returns

14 The **fwprintf** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

Environmental limits

15 The number of wide characters that can be produced by any single conversion shall be at least 4095.

16 EXAMPLE To print a date and time in the form ‘‘Sunday, July 3, 10:02’’ followed by π to five decimal places:

```
#include <math.h>
#include <stdio.h>
#include <wchar.h>
/* ... */
wchar_t *weekday, *month;   // pointers to wide strings
int day, hour, min;
fwprintf(stdout, L"%ls, %ls %d, %.2d:%.2d\n",
      weekday, month, day, hour, min);
fwprintf(stdout, L"pi = %.5f\n", 4 * atan(1.0));
```

Forward references: the **btowc** function (7.29.6.1.1), the **mbrtowc** function (7.29.6.3.2).

7.29.2.2 The fwscanf function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
int fwscanf(FILE * restrict stream,
     const wchar_t * restrict format, ...);
```

Description

2 The **fwscanf** function reads input from the stream pointed to by **stream**, under control of the wide string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments

336) For binary-to-decimal conversion, the result format’s values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

3 The format is composed of zero or more directives: one or more white-space wide characters, an ordinary wide character (neither **%** nor a white-space wide character), or a conversion specification. Each conversion specification is introduced by the wide character **%**. After the **%**, the following appear in sequence:

— An optional assignment-suppressing wide character *****.

— An optional decimal integer greater than zero that specifies the maximum field width (in wide characters).

— An optional *length modifier* that specifies the size of the receiving object.

— A *conversion specifier* wide character that specifies the type of conversion to be applied.

4 The **fwscanf** function executes each directive of the format in turn. When all directives have been executed, or if a directive fails (as detailed below), the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

5 A directive composed of white-space wide character(s) is executed by reading input up to the first non-white-space wide character (which remains unread), or until no more wide characters can be read. The directive never fails.

6 A directive that is an ordinary wide character is executed by reading the next wide character of the stream. If that wide character differs from the directive, the directive fails and the differing and subsequent wide characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a wide character from being read, the directive fails.

7 A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

8 Input white-space wide characters (as specified by the **iswspace** function) are skipped, unless the specification includes a **[**, **c**, or **n** specifier.[337)]

9 An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest sequence of input wide characters which does not exceed any specified field width and which is, or is a prefix of, a matching input

337) These white-space wide characters are not counted against a specified field width.

sequence.[338] The first wide character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

10 Except in the case of a **%** specifier, the input item (or, in the case of a **%n** directive, the count of input wide characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *****, the result of the conversion is placed in the object pointed to by the first argument following the **format** argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

11 The length modifiers and their meanings are:

hh Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.

h Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **short int** or **unsigned short int**.

l (ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **long int** or **unsigned long int**; that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to an argument with type pointer to **double**; or that a following **c**, **s**, or **[** conversion specifier applies to an argument with type pointer to **wchar_t**.

ll (ell-ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **long long int** or **unsigned long long int**.

j Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **intmax_t** or **uintmax_t**.

z Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **size_t** or the corresponding signed integer type.

t Specifies that a following **d**, **i**, **o**, **u**, **x**, **X**, or **n** conversion specifier applies to an argument with type pointer to **ptrdiff_t** or the corresponding unsigned integer type.

338) **fwscanf** pushes back at most one input wide character onto the input stream. Therefore, some sequences that are acceptable to **wcstod**, **wcstol**, etc., are unacceptable to **fwscanf**.

L Specifies that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to an argument with type pointer to **long double**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

12 The conversion specifiers and their meanings are:

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **wcstol** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to signed integer.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **wcstol** function with the value 0 for the **base** argument. The corresponding argument shall be a pointer to signed integer.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **wcstoul** function with the value 8 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **wcstoul** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **wcstoul** function with the value 16 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

a,e,f,g Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the **wcstod** function. The corresponding argument shall be a pointer to floating.

c Matches a sequence of wide characters of exactly the number specified by the field width (1 if no field width is present in the directive).

If no **l** length modifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence. No null wide character is added.

s Matches a sequence of non-white-space wide characters.

If no **l** length modifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

[Matches a nonempty sequence of wide characters from a set of expected characters (the *scanset*).

If no **l** length modifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent wide characters in the **format** string, up to and including the matching right bracket (**]**). The wide characters between the brackets (the *scanlist*) compose the scanset, unless the wide character after the left bracket is a circumflex (**^**), in which case the scanset contains all wide characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with **[]** or **[^]**, the right bracket wide character is in the scanlist and the next following right bracket wide character is the matching right bracket that ends the specification; otherwise the first following right bracket wide character is the one that ends the specification. If a **-** wide character is in the scanlist and is not the first, nor the second where the first wide character is a **^**, nor the last character, the behavior is implementation-defined.

p Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the **%p** conversion of the **fwprintf** function. The corresponding argument shall be a pointer to a pointer to **void**. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the **%p** conversion is undefined.

n No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of wide characters read from the input stream so far by this call to the **fwscanf** function. Execution of a **%n** directive does not increment the assignment count returned at the completion of execution of the **fwscanf** function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing wide character or a field width, the behavior is undefined.

% Matches a single **%** wide character; no conversion or assignment occurs. The complete conversion specification shall be **%%**.

13 If a conversion specification is invalid, the behavior is undefined.[339)]

14 The conversion specifiers **A**, **E**, **F**, **G**, and **X** are also valid and behave the same as, respectively, **a**, **e**, **f**, **g**, and **x**.

15 Trailing white space (including new-line wide characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** directive.

Returns

16 The **fwscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

17 EXAMPLE 1 The call:

```
#include <stdio.h>
#include <wchar.h>
/* ... */
int n, i; float x; wchar_t name[50];
n = fwscanf(stdin, L"%d%f%ls", &i, &x, name);
```

339) See ‘‘future library directions’’ (7.31.16).

with the input line:

```
25 54.32E-1 thompson
```

will assign to `n` the value `3`, to `i` the value `25`, to `x` the value `5.432`, and to `name` the sequence `thompson\0`.

18 EXAMPLE 2 The call:

```
#include <stdio.h>
#include <wchar.h>
/* ... */
int i; float x; double y;
fwscanf(stdin, L"%2d%f%*d %lf", &i, &x, &y);
```

with input:

```
56789 0123 56a72
```

will assign to `i` the value `56` and to `x` the value `789.0`, will skip past `0123`, and will assign to `y` the value `56.0`. The next wide character read from the input stream will be `a`.

Forward references: the `wcstod`, `wcstof`, and `wcstold` functions (7.29.4.1.1), the `wcstol`, `wcstoll`, `wcstoul`, and `wcstoull` functions (7.29.4.1.2), the `wcrtomb` function (7.29.6.3.3).

7.29.2.3 The `swprintf` function

Synopsis

1

```
#include <wchar.h>
int swprintf(wchar_t * restrict s,
     size_t n,
     const wchar_t * restrict format, ...);
```

Description

2 The `swprintf` function is equivalent to `fwprintf`, except that the argument `s` specifies an array of wide characters into which the generated output is to be written, rather than written to a stream. No more than `n` wide characters are written, including a terminating null wide character, which is always added (unless `n` is zero).

Returns

3 The `swprintf` function returns the number of wide characters written in the array, not counting the terminating null wide character, or a negative value if an encoding error occurred or if `n` or more wide characters were requested to be written.

7.29.2.4 The `swscanf` function

Synopsis

1
```
#include <wchar.h>
int swscanf(const wchar_t * restrict s,
     const wchar_t * restrict format, ...);
```

Description

2 The **swscanf** function is equivalent to **fwscanf**, except that the argument **s** specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for the **fwscanf** function.

Returns

3 The **swscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **swscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.2.5 The `vfwprintf` function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf(FILE * restrict stream,
     const wchar_t * restrict format,
     va_list arg);
```

Description

2 The **vfwprintf** function is equivalent to **fwprintf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfwprintf** function does not invoke the **va_end** macro.[340)]

Returns

3 The **vfwprintf** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

---

340) As the functions **vfwprintf**, **vswprintf**, **vfwscanf**, **vwprintf**, **vwscanf**, and **vswscanf** invoke the **va_arg** macro, the value of **arg** after the return is indeterminate.

4 EXAMPLE The following shows the use of the **vfwprintf** function in a general error-reporting routine.

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

void error(char *function_name, wchar_t *format, ...)
{
      va_list args;

      va_start(args, format);
      // print out name of function causing error
      fwprintf(stderr, L"ERROR in %s: ", function_name);
      // print out remainder of message
      vfwprintf(stderr, format, args);
      va_end(args);
}
```

7.29.2.6 The vfwscanf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwscanf(FILE * restrict stream,
     const wchar_t * restrict format,
     va_list arg);
```

Description

2 The **vfwscanf** function is equivalent to **fwscanf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfwscanf** function does not invoke the **va_end** macro.[340)]

Returns

3 The **vfwscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vfwscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.2.7 The `vswprintf` function

Synopsis

1
```
#include <stdarg.h>
#include <wchar.h>
int vswprintf(wchar_t * restrict s,
     size_t n,
     const wchar_t * restrict format,
     va_list arg);
```

Description

2 The **`vswprintf`** function is equivalent to **`swprintf`**, with the variable argument list replaced by **`arg`**, which shall have been initialized by the **`va_start`** macro (and possibly subsequent **`va_arg`** calls). The **`vswprintf`** function does not invoke the **`va_end`** macro.[340)]

Returns

3 The **`vswprintf`** function returns the number of wide characters written in the array, not counting the terminating null wide character, or a negative value if an encoding error occurred or if **`n`** or more wide characters were requested to be generated.

7.29.2.8 The `vswscanf` function

Synopsis

1
```
#include <stdarg.h>
#include <wchar.h>
int vswscanf(const wchar_t * restrict s,
     const wchar_t * restrict format,
     va_list arg);
```

Description

2 The **`vswscanf`** function is equivalent to **`swscanf`**, with the variable argument list replaced by **`arg`**, which shall have been initialized by the **`va_start`** macro (and possibly subsequent **`va_arg`** calls). The **`vswscanf`** function does not invoke the **`va_end`** macro.[340)]

Returns

3 The **`vswscanf`** function returns the value of the macro **`EOF`** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **`vswscanf`** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.2.9 The `vwprintf` function

Synopsis

1
```
#include <stdarg.h>
#include <wchar.h>
int vwprintf(const wchar_t * restrict format,
     va_list arg);
```

Description

2 The **`vwprintf`** function is equivalent to **`wprintf`**, with the variable argument list replaced by **`arg`**, which shall have been initialized by the **`va_start`** macro (and possibly subsequent **`va_arg`** calls). The **`vwprintf`** function does not invoke the **`va_end`** macro.[340)]

Returns

3 The **`vwprintf`** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

7.29.2.10 The `vwscanf` function

Synopsis

1
```
#include <stdarg.h>
#include <wchar.h>
int vwscanf(const wchar_t * restrict format,
     va_list arg);
```

Description

2 The **`vwscanf`** function is equivalent to **`wscanf`**, with the variable argument list replaced by **`arg`**, which shall have been initialized by the **`va_start`** macro (and possibly subsequent **`va_arg`** calls). The **`vwscanf`** function does not invoke the **`va_end`** macro.[340)]

Returns

3 The **`vwscanf`** function returns the value of the macro **`EOF`** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **`vwscanf`** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.2.11 The `wprintf` function

Synopsis

1
```
#include <wchar.h>
int wprintf(const wchar_t * restrict format, ...);
```

Description

2 The **`wprintf`** function is equivalent to **`fwprintf`** with the argument **`stdout`** interposed before the arguments to **`wprintf`**.

Returns

3 The **`wprintf`** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

7.29.2.12 The `wscanf` function

Synopsis

1
```
#include <wchar.h>
int wscanf(const wchar_t * restrict format, ...);
```

Description

2 The **`wscanf`** function is equivalent to **`fwscanf`** with the argument **`stdin`** interposed before the arguments to **`wscanf`**.

Returns

3 The **`wscanf`** function returns the value of the macro **`EOF`** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **`wscanf`** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.3 Wide character input/output functions

7.29.3.1 The `fgetwc` function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *stream);
```

Description

2 If the end-of-file indicator for the input stream pointed to by **`stream`** is not set and a next wide character is present, the **`fgetwc`** function obtains that wide character as a **`wchar_t`** converted to a **`wint_t`** and advances the associated file position indicator for the stream (if defined).

Returns

3 If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream is set and the **fgetwc** function returns **WEOF**. Otherwise, the **fgetwc** function returns the next wide character from the input stream pointed to by **stream**. If a read error occurs, the error indicator for the stream is set and the **fgetwc** function returns **WEOF**. If an encoding error occurs (including too few bytes), the value of the macro **EILSEQ** is stored in **errno** and the **fgetwc** function returns **WEOF**.[341)]

7.29.3.2 The fgetws function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t * restrict s,
     int n, FILE * restrict stream);
```

Description

2 The **fgetws** function reads at most one less than the number of wide characters specified by **n** from the stream pointed to by **stream** into the array pointed to by **s**. No additional wide characters are read after a new-line wide character (which is retained) or after end-of-file. A null wide character is written immediately after the last wide character read into the array.

Returns

3 The **fgetws** function returns **s** if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read or encoding error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

7.29.3.3 The fputwc function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t fputwc(wchar_t c, FILE *stream);
```

Description

2 The **fputwc** function writes the wide character specified by **c** to the output stream pointed to by **stream**, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot

341) An end-of-file and a read error can be distinguished by use of the **feof** and **ferror** functions. Also, **errno** will be set to **EILSEQ** by input/output functions only if an encoding error occurs.

support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Returns

3 The **fputwc** function returns the wide character written. If a write error occurs, the error indicator for the stream is set and **fputwc** returns **WEOF**. If an encoding error occurs, the value of the macro **EILSEQ** is stored in **errno** and **fputwc** returns **WEOF**.

7.29.3.4 The fputws function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
int fputws(const wchar_t * restrict s,
     FILE * restrict stream);
```

Description

2 The **fputws** function writes the wide string pointed to by **s** to the stream pointed to by **stream**. The terminating null wide character is not written.

Returns

3 The **fputws** function returns **EOF** if a write or encoding error occurs; otherwise, it returns a nonnegative value.

7.29.3.5 The fwide function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
int fwide(FILE *stream, int mode);
```

Description

2 The **fwide** function determines the orientation of the stream pointed to by **stream**. If **mode** is greater than zero, the function first attempts to make the stream wide oriented. If **mode** is less than zero, the function first attempts to make the stream byte oriented.[342] Otherwise, **mode** is zero and the function does not alter the orientation of the stream.

Returns

3 The **fwide** function returns a value greater than zero if, after the call, the stream has wide orientation, a value less than zero if the stream has byte orientation, or zero if the stream has no orientation.

---

342) If the orientation of the stream has already been determined, **fwide** does not change it.

7.29.3.6 The `getwc` function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *stream);
```

Description

2 The **getwc** function is equivalent to **fgetwc**, except that if it is implemented as a macro, it may evaluate **stream** more than once, so the argument should never be an expression with side effects.

Returns

3 The **getwc** function returns the next wide character from the input stream pointed to by **stream**, or **WEOF**.

7.29.3.7 The `getwchar` function

Synopsis

1
```
#include <wchar.h>
wint_t getwchar(void);
```

Description

2 The **getwchar** function is equivalent to **getwc** with the argument **stdin**.

Returns

3 The **getwchar** function returns the next wide character from the input stream pointed to by **stdin**, or **WEOF**.

7.29.3.8 The `putwc` function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wchar_t c, FILE *stream);
```

Description

2 The **putwc** function is equivalent to **fputwc**, except that if it is implemented as a macro, it may evaluate **stream** more than once, so that argument should never be an expression with side effects.

Returns

3 The **putwc** function returns the wide character written, or **WEOF**.

7.29.3.9 The `putwchar` function

Synopsis

1
```
#include <wchar.h>
wint_t putwchar(wchar_t c);
```

Description

2 The **putwchar** function is equivalent to **putwc** with the second argument **stdout**.

Returns

3 The **putwchar** function returns the character written, or **WEOF**.

7.29.3.10 The `ungetwc` function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t ungetwc(wint_t c, FILE *stream);
```

Description

2 The **ungetwc** function pushes the wide character specified by **c** back onto the input stream pointed to by **stream**. Pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by **stream**) to a file positioning function (**fseek**, **fsetpos**, or **rewind**) discards any pushed-back wide characters for the stream. The external storage corresponding to the stream is unchanged.

3 One wide character of pushback is guaranteed, even if the call to the **ungetwc** function follows just after a call to a formatted wide character input function **fwscanf**, **vfwscanf**, **vwscanf**, or **wscanf**. If the **ungetwc** function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

4 If the value of **c** equals that of the macro **WEOF**, the operation fails and the input stream is unchanged.

5 A successful call to the **ungetwc** function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back wide characters is the same as it was before the wide characters were pushed back. For a text or binary stream, the value of its file position indicator after a successful call to the **ungetwc** function is unspecified until all pushed-back wide characters are read or discarded.

Returns

6 The **ungetwc** function returns the wide character pushed back, or **WEOF** if the operation fails.

7.29.4 General wide string utilities

1 The header **<wchar.h>** declares a number of functions useful for wide string manipulation. Various methods are used for determining the lengths of the arrays, but in all cases a **wchar_t *** argument points to the initial (lowest addressed) element of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

2 Where an argument declared as **size_t n** determines the length of the array for a function, **n** can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid values, as described in 7.1.4. On such a call, a function that locates a wide character finds no occurrence, a function that compares two wide character sequences returns zero, and a function that copies wide characters copies zero wide characters.

7.29.4.1 Wide string numeric conversion functions

7.29.4.1.1 The wcstod, wcstof, and wcstold functions

Synopsis

1
```
#include <wchar.h>
double wcstod(const wchar_t * restrict nptr,
     wchar_t ** restrict endptr);
float wcstof(const wchar_t * restrict nptr,
     wchar_t ** restrict endptr);
long double wcstold(const wchar_t * restrict nptr,
     wchar_t ** restrict endptr);
```

Description

2 The **wcstod**, **wcstof**, and **wcstold** functions convert the initial portion of the wide string pointed to by **nptr** to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters (as specified by the **iswspace** function), a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

3 The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

— a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part as defined for the corresponding single-byte characters in 6.4.4.2;

— a **`0x`** or **`0X`**, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point wide character, then an optional binary exponent part as defined in 6.4.4.2;

— **`INF`** or **`INFINITY`**, or any other wide string equivalent except for case

— **`NAN`** or **`NAN(`***n-wchar-sequence*$_{opt}$**`)`**, or any other wide string equivalent except for case in the **`NAN`** part, where:

n-wchar-sequence:
digit
nondigit
n-wchar-sequence digit
n-wchar-sequence nondigit

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

4 If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, except that the decimal-point wide character is used in place of a period, and that if neither an exponent part nor a decimal-point wide character appears in a decimal floating point number, or if a binary exponent part does not appear in a hexadecimal floating point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated.[343] A wide character sequence **`INF`** or **`INFINITY`** is interpreted as an infinity, if representable in the return type, else like a floating constant that is too large for the range of the return type. A wide character sequence **`NAN`** or **`NAN(`***n-wchar-sequence*$_{opt}$**`)`** is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the meaning of the n-wchar sequence is implementation-defined.[344] A pointer to the

343) It is unspecified whether a minus-signed sequence is converted to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (see F.5); the two methods may yield different results if rounding is toward positive or negative infinity. In either case, the functions honor the sign of zero if floating-point arithmetic supports signed zeros.

344) An implementation may use the n-wchar sequence to determine extra information to be represented in the NaN's significand.

final wide string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

5 If the subject sequence has the hexadecimal form and **FLT_RADIX** is a power of 2, the value resulting from the conversion is correctly rounded.

6 In other than the **"C"** locale, additional locale-specific subject sequence forms may be accepted.

7 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Recommended practice

8 If the subject sequence has the hexadecimal form, **FLT_RADIX** is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

9 If the subject sequence has the decimal form and at most **DECIMAL_DIG** (defined in **<float.h>**) significant digits, the result should be correctly rounded. If the subject sequence D has the decimal form and more than **DECIMAL_DIG** significant digits, consider the two bounding, adjacent decimal strings L and U, both having **DECIMAL_DIG** significant digits, such that the values of L, D, and U satisfy $L \leq D \leq U$. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D should have a correct sign for the current rounding direction.[345)]

Returns

10 The functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value overflows and default rounding is in effect (7.12.1), plus or minus **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** is returned (according to the return type and sign of the value), and the value of the macro **ERANGE** is stored in **errno**. If the result underflows (7.12.1), the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether **errno** acquires the value **ERANGE** is implementation-defined.

345) **DECIMAL_DIG**, defined in **<float.h>**, should be sufficiently large that L and U will usually round to the same internal floating value, but if not will round to adjacent values.

7.29.4.1.2 The `wcstol`, `wcstoll`, `wcstoul`, and `wcstoull` functions

Synopsis

1
```
#include <wchar.h>
long int wcstol(
     const wchar_t * restrict nptr,
     wchar_t ** restrict endptr,
     int base);
long long int wcstoll(
     const wchar_t * restrict nptr,
     wchar_t ** restrict endptr,
     int base);
unsigned long int wcstoul(
     const wchar_t * restrict nptr,
     wchar_t ** restrict endptr,
     int base);
unsigned long long int wcstoull(
     const wchar_t * restrict nptr,
     wchar_t ** restrict endptr,
     int base);
```

Description

2 The **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** functions convert the initial portion of the wide string pointed to by **nptr** to **long int**, **long long int**, **unsigned long int**, and **unsigned long long int** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters (as specified by the **iswspace** function), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to an integer, and return the result.

3 If the value of **base** is zero, the expected form of the subject sequence is that of an integer constant as described for the corresponding single-byte characters in 6.4.4.1, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of **base** is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by **base**, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from **a** (or **A**) through **z** (or **Z**) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of **base** are permitted. If the value of **base** is 16, the wide characters **0x** or **0X** may optionally precede the sequence of letters and digits, following the sign if present.

4 The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is empty or consists entirely of white space, or if the first non-white-space wide character is other than a sign or a permissible letter or digit.

5 If the subject sequence has the expected form and the value of **base** is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of 6.4.4.1. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated (in the return type). A pointer to the final wide string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

6 In other than the **"C"** locale, additional locale-specific subject sequence forms may be accepted.

7 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Returns

8 The **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG_MIN**, **LONG_MAX**, **LLONG_MIN**, **LLONG_MAX**, **ULONG_MAX**, or **ULLONG_MAX** is returned (according to the return type sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

7.29.4.2 Wide string copying functions

7.29.4.2.1 The wcscpy function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcscpy(wchar_t * restrict s1,
     const wchar_t * restrict s2);
```

Description

2 The **wcscpy** function copies the wide string pointed to by **s2** (including the terminating null wide character) into the array pointed to by **s1**.

Returns

3 The **wcscpy** function returns the value of **s1**.

7.29.4.2.2 The `wcsncpy` function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcsncpy(wchar_t * restrict s1,
     const wchar_t * restrict s2,
     size_t n);
```

Description

2 The **wcsncpy** function copies not more than **n** wide characters (those that follow a null wide character are not copied) from the array pointed to by **s2** to the array pointed to by **s1**.[346)]

3 If the array pointed to by **s2** is a wide string that is shorter than **n** wide characters, null wide characters are appended to the copy in the array pointed to by **s1**, until **n** wide characters in all have been written.

Returns

4 The **wcsncpy** function returns the value of **s1**.

7.29.4.2.3 The `wmemcpy` function

Synopsis

1
```
#include <wchar.h>
wchar_t *wmemcpy(wchar_t * restrict s1,
     const wchar_t * restrict s2,
     size_t n);
```

Description

2 The **wmemcpy** function copies **n** wide characters from the object pointed to by **s2** to the object pointed to by **s1**.

Returns

3 The **wmemcpy** function returns the value of **s1**.

346) Thus, if there is no null wide character in the first **n** wide characters of the array pointed to by **s2**, the result will not be null-terminated.

7.29.4.2.4 The `wmemmove` function

Synopsis

1
```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2,
     size_t n);
```

Description

2 The **wmemmove** function copies **n** wide characters from the object pointed to by **s2** to the object pointed to by **s1**. Copying takes place as if the **n** wide characters from the object pointed to by **s2** are first copied into a temporary array of **n** wide characters that does not overlap the objects pointed to by **s1** or **s2**, and then the **n** wide characters from the temporary array are copied into the object pointed to by **s1**.

Returns

3 The **wmemmove** function returns the value of **s1**.

7.29.4.3 Wide string concatenation functions

7.29.4.3.1 The `wcscat` function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcscat(wchar_t * restrict s1,
     const wchar_t * restrict s2);
```

Description

2 The **wcscat** function appends a copy of the wide string pointed to by **s2** (including the terminating null wide character) to the end of the wide string pointed to by **s1**. The initial wide character of **s2** overwrites the null wide character at the end of **s1**.

Returns

3 The **wcscat** function returns the value of **s1**.

7.29.4.3.2 The `wcsncat` function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcsncat(wchar_t * restrict s1,
     const wchar_t * restrict s2,
     size_t n);
```

Description

2 The **wcsncat** function appends not more than **n** wide characters (a null wide character and those that follow it are not appended) from the array pointed to by **s2** to the end of

the wide string pointed to by `s1`. The initial wide character of `s2` overwrites the null wide character at the end of `s1`. A terminating null wide character is always appended to the result.[347)]

Returns

3 The `wcsncat` function returns the value of `s1`.

7.29.4.4 Wide string comparison functions

1 Unless explicitly stated otherwise, the functions described in this subclause order two wide characters the same way as two integers of the underlying integer type designated by `wchar_t`.

7.29.4.4.1 The `wcscmp` function

Synopsis

1
```
#include <wchar.h>
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The `wcscmp` function compares the wide string pointed to by `s1` to the wide string pointed to by `s2`.

Returns

3 The `wcscmp` function returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by `s1` is greater than, equal to, or less than the wide string pointed to by `s2`.

7.29.4.4.2 The `wcscoll` function

Synopsis

1
```
#include <wchar.h>
int wcscoll(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The `wcscoll` function compares the wide string pointed to by `s1` to the wide string pointed to by `s2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale.

Returns

3 The `wcscoll` function returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by `s1` is greater than, equal to, or less than the

347) Thus, the maximum number of wide characters that can end up in the array pointed to by `s1` is `wcslen(s1)+n+1`.

wide string pointed to by **s2** when both are interpreted as appropriate to the current locale.

7.29.4.4.3 The `wcsncmp` function

Synopsis

1
```
#include <wchar.h>
int wcsncmp(const wchar_t *s1, const wchar_t *s2,
     size_t n);
```

Description

2 The **wcsncmp** function compares not more than **n** wide characters (those that follow a null wide character are not compared) from the array pointed to by **s1** to the array pointed to by **s2**.

Returns

3 The **wcsncmp** function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by **s1** is greater than, equal to, or less than the possibly null-terminated array pointed to by **s2**.

7.29.4.4.4 The `wcsxfrm` function

Synopsis

1
```
#include <wchar.h>
size_t wcsxfrm(wchar_t * restrict s1,
     const wchar_t * restrict s2,
     size_t n);
```

Description

2 The **wcsxfrm** function transforms the wide string pointed to by **s2** and places the resulting wide string into the array pointed to by **s1**. The transformation is such that if the **wcscmp** function is applied to two transformed wide strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **wcscoll** function applied to the same two original wide strings. No more than **n** wide characters are placed into the resulting array pointed to by **s1**, including the terminating null wide character. If **n** is zero, **s1** is permitted to be a null pointer.

Returns

3 The **wcsxfrm** function returns the length of the transformed wide string (not including the terminating null wide character). If the value returned is **n** or greater, the contents of the array pointed to by **s1** are indeterminate.

4 EXAMPLE The value of the following expression is the length of the array needed to hold the transformation of the wide string pointed to by **s**:

```
1 + wcsxfrm(NULL, s, 0)
```

7.29.4.4.5 The `wmemcmp` function

Synopsis

1
```
#include <wchar.h>
int wmemcmp(const wchar_t *s1, const wchar_t *s2,
     size_t n);
```

Description

2 The **`wmemcmp`** function compares the first **`n`** wide characters of the object pointed to by **`s1`** to the first **`n`** wide characters of the object pointed to by **`s2`**.

Returns

3 The **`wmemcmp`** function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by **`s1`** is greater than, equal to, or less than the object pointed to by **`s2`**.

7.29.4.5 Wide string search functions

7.29.4.5.1 The `wcschr` function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

Description

2 The **`wcschr`** function locates the first occurrence of **`c`** in the wide string pointed to by **`s`**. The terminating null wide character is considered to be part of the wide string.

Returns

3 The **`wcschr`** function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the wide string.

7.29.4.5.2 The `wcscspn` function

Synopsis

1
```
#include <wchar.h>
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The **`wcscspn`** function computes the length of the maximum initial segment of the wide string pointed to by **`s1`** which consists entirely of wide characters *not* from the wide string pointed to by **`s2`**.

Returns

3 The **wcscspn** function returns the length of the segment.

7.29.4.5.3 The wcspbrk function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The **wcspbrk** function locates the first occurrence in the wide string pointed to by **s1** of any wide character from the wide string pointed to by **s2**.

Returns

3 The **wcspbrk** function returns a pointer to the wide character in **s1**, or a null pointer if no wide character from **s2** occurs in **s1**.

7.29.4.5.4 The wcsrchr function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
```

Description

2 The **wcsrchr** function locates the last occurrence of **c** in the wide string pointed to by **s**. The terminating null wide character is considered to be part of the wide string.

Returns

3 The **wcsrchr** function returns a pointer to the wide character, or a null pointer if **c** does not occur in the wide string.

7.29.4.5.5 The wcsspn function

Synopsis

1
```
#include <wchar.h>
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The **wcsspn** function computes the length of the maximum initial segment of the wide string pointed to by **s1** which consists entirely of wide characters from the wide string pointed to by **s2**.

Returns

3 The **wcsspn** function returns the length of the segment.

7.29.4.5.6 The `wcsstr` function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The **wcsstr** function locates the first occurrence in the wide string pointed to by **s1** of the sequence of wide characters (excluding the terminating null wide character) in the wide string pointed to by **s2**.

Returns

3 The **wcsstr** function returns a pointer to the located wide string, or a null pointer if the wide string is not found. If **s2** points to a wide string with zero length, the function returns **s1**.

7.29.4.5.7 The `wcstok` function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcstok(wchar_t * restrict s1,
     const wchar_t * restrict s2,
     wchar_t ** restrict ptr);
```

Description

2 A sequence of calls to the **wcstok** function breaks the wide string pointed to by **s1** into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by **s2**. The third argument points to a caller-provided **wchar_t** pointer into which the **wcstok** function stores information necessary for it to continue scanning the same wide string.

3 The first call in a sequence has a non-null first argument and stores an initial value in the object pointed to by **ptr**. Subsequent calls in the sequence have a null first argument and the object pointed to by **ptr** is required to have the value stored by the previous call in the sequence, which is then updated. The separator wide string pointed to by **s2** may be different from call to call.

4 The first call in the sequence searches the wide string pointed to by **s1** for the first wide character that is *not* contained in the current separator wide string pointed to by **s2**. If no such wide character is found, then there are no tokens in the wide string pointed to by **s1** and the **wcstok** function returns a null pointer. If such a wide character is found, it is the start of the first token.

5 The **wcstok** function then searches from there for a wide character that *is* contained in the current separator wide string. If no such wide character is found, the current token

extends to the end of the wide string pointed to by **s1**, and subsequent searches in the same wide string for a token return a null pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token.

6 In all cases, the **wcstok** function stores sufficient information in the pointer pointed to by **ptr** so that subsequent calls, with a null pointer for **s1** and the unmodified pointer value for **ptr**, shall start searching just past the element overwritten by a null wide character (if any).

Returns

7 The **wcstok** function returns a pointer to the first wide character of a token, or a null pointer if there is no token.

8 EXAMPLE

```
#include <wchar.h>
static wchar_t str1[] = L"?a???b,,,#c";
static wchar_t str2[] = L"\t \t";
wchar_t *t, *ptr1, *ptr2;

t = wcstok(str1, L"?", &ptr1);       // t points to the token L"a"
t = wcstok(NULL, L",", &ptr1);       // t points to the token L"??b"
t = wcstok(str2, L" \t", &ptr2);     // t is a null pointer
t = wcstok(NULL, L"#,", &ptr1);      // t points to the token L"c"
t = wcstok(NULL, L"?", &ptr1);       // t is a null pointer
```

7.29.4.5.8 The wmemchr function

Synopsis

1
```
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *s, wchar_t c,
     size_t n);
```

Description

2 The **wmemchr** function locates the first occurrence of **c** in the initial **n** wide characters of the object pointed to by **s**.

Returns

3 The **wmemchr** function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the object.

7.29.4.6 Miscellaneous functions

7.29.4.6.1 The `wcslen` function

Synopsis

1

```
#include <wchar.h>
size_t wcslen(const wchar_t *s);
```

Description

2 The **`wcslen`** function computes the length of the wide string pointed to by **`s`**.

Returns

3 The **`wcslen`** function returns the number of wide characters that precede the terminating null wide character.

7.29.4.6.2 The `wmemset` function

Synopsis

1

```
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

Description

2 The **`wmemset`** function copies the value of **`c`** into each of the first **`n`** wide characters of the object pointed to by **`s`**.

Returns

3 The **`wmemset`** function returns the value of **`s`**.

7.29.5 Wide character time conversion functions

7.29.5.1 The `wcsftime` function

Synopsis

1

```
#include <time.h>
#include <wchar.h>
size_t wcsftime(wchar_t * restrict s,
     size_t maxsize,
     const wchar_t * restrict format,
     const struct tm * restrict timeptr);
```

Description

2 The **`wcsftime`** function is equivalent to the **`strftime`** function, except that:

— The argument **`s`** points to the initial element of an array of wide characters into which the generated output is to be placed.

— The argument **maxsize** indicates the limiting number of wide characters.

— The argument **format** is a wide string and the conversion specifiers are replaced by corresponding sequences of wide characters.

— The return value indicates the number of wide characters.

Returns

3 If the total number of resulting wide characters including the terminating null wide character is not more than **maxsize**, the **wcsftime** function returns the number of wide characters placed into the array pointed to by **s** not including the terminating null wide character. Otherwise, zero is returned and the contents of the array are indeterminate.

7.29.6 Extended multibyte/wide character conversion utilities

1 The header **<wchar.h>** declares an extended set of functions useful for conversion between multibyte characters and wide characters.

2 Most of the following functions — those that are listed as ‘‘restartable’’, 7.29.6.3 and 7.29.6.4 — take as a last argument a pointer to an object of type **mbstate_t** that is used to describe the current *conversion state* from a particular multibyte character sequence to a wide character sequence (or the reverse) under the rules of a particular setting for the **LC_CTYPE** category of the current locale.

3 The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new multibyte character in the initial shift state. A zero-valued **mbstate_t** object is (at least) one way to describe an initial conversion state. A zero-valued **mbstate_t** object can be used to initiate conversion involving any multibyte character sequence, in any **LC_CTYPE** category setting. If an **mbstate_t** object has been altered by any of the functions described in this subclause, and is then used with a different multibyte character sequence, or in the other conversion direction, or with a different **LC_CTYPE** category setting than on earlier function calls, the behavior is undefined.[348)]

4 On entry, each function takes the described conversion state (either internal or pointed to by an argument) as current. The conversion state described by the referenced object is altered as needed to track the shift state, and the position within a multibyte character, for the associated multibyte character sequence.

348) Thus, a particular **mbstate_t** object can be used, for example, with both the **mbrtowc** and **mbsrtowcs** functions as long as they are used to step sequentially through the same multibyte character string.

7.29.6.1 Single-byte/wide character conversion functions

7.29.6.1.1 The `btowc` function

Synopsis

1
```
#include <wchar.h>
wint_t btowc(int c);
```

Description

2 The `btowc` function determines whether `c` constitutes a valid single-byte character in the initial shift state.

Returns

3 The `btowc` function returns `WEOF` if `c` has the value `EOF` or if `(unsigned char)c` does not constitute a valid single-byte character in the initial shift state. Otherwise, it returns the wide character representation of that character.

7.29.6.1.2 The `wctob` function

Synopsis

1
```
#include <wchar.h>
int wctob(wint_t c);
```

Description

2 The `wctob` function determines whether `c` corresponds to a member of the extended character set whose multibyte character representation is a single byte when in the initial shift state.

Returns

3 The `wctob` function returns `EOF` if `c` does not correspond to a multibyte character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character as an `unsigned char` converted to an `int`.

7.29.6.2 Conversion state functions

7.29.6.2.1 The `mbsinit` function

Synopsis

1
```
#include <wchar.h>
int mbsinit(const mbstate_t *ps);
```

Description

2 If `ps` is not a null pointer, the `mbsinit` function determines whether the referenced `mbstate_t` object describes an initial conversion state.

Returns

3 The **mbsinit** function returns nonzero if **ps** is a null pointer or if the referenced object describes an initial conversion state; otherwise, it returns zero.

7.29.6.3 Restartable multibyte/wide character conversion functions

1 These functions differ from the corresponding multibyte character functions of 7.22.7 (**mblen**, **mbtowc**, and **wctomb**) in that they have an extra parameter, **ps**, of type pointer to **mbstate_t** that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If **ps** is a null pointer, each function uses its own internal **mbstate_t** object instead, which is initialized at program startup to the initial conversion state; the functions are not required to avoid data races with other calls to the same function in this case. The implementation behaves as if no library function calls these functions with a null pointer for **ps**.

2 Also unlike their corresponding functions, the return value does not represent whether the encoding is state-dependent.

7.29.6.3.1 The mbrlen function

Synopsis

1
```
#include <wchar.h>
size_t mbrlen(const char * restrict s,
     size_t n,
     mbstate_t * restrict ps);
```

Description

2 The **mbrlen** function is equivalent to the call:

```
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal)
```

where **internal** is the **mbstate_t** object for the **mbrlen** function, except that the expression designated by **ps** is evaluated only once.

Returns

3 The **mbrlen** function returns a value between zero and **n**, inclusive, **(size_t)(-2)**, or **(size_t)(-1)**.

Forward references: the **mbrtowc** function (7.29.6.3.2).

7.29.6.3.2 The `mbrtowc` function

Synopsis

1
```
#include <wchar.h>
size_t mbrtowc(wchar_t * restrict pwc,
     const char * restrict s,
     size_t n,
     mbstate_t * restrict ps);
```

Description

2 If `s` is a null pointer, the `mbrtowc` function is equivalent to the call:

```
mbrtowc(NULL, "", 1, ps)
```

In this case, the values of the parameters `pwc` and `n` are ignored.

3 If `s` is not a null pointer, the `mbrtowc` function inspects at most `n` bytes beginning with the byte pointed to by `s` to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the value of the corresponding wide character and then, if `pwc` is not a null pointer, stores that value in the object pointed to by `pwc`. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

4 The `mbrtowc` function returns the first of the following that applies (given the current conversion state):

0 — if the next `n` or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 *and* `n` *inclusive* — if the next `n` or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

`(size_t)(-2)` — if the next `n` bytes contribute to an incomplete (but potentially valid) multibyte character, and all `n` bytes have been processed (no value is stored).[349)]

`(size_t)(-1)` — if an encoding error occurs, in which case the next `n` or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro `EILSEQ` is stored in `errno`, and the conversion state is unspecified.

---

349) When `n` has at least the value of the `MB_CUR_MAX` macro, this case can only occur if `s` points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

7.29.6.3.3 The `wcrtomb` function

Synopsis

1
```
#include <wchar.h>
size_t wcrtomb(char * restrict s,
     wchar_t wc,
     mbstate_t * restrict ps);
```

Description

2 If **s** is a null pointer, the **wcrtomb** function is equivalent to the call

```
wcrtomb(buf, L'\0', ps)
```

where **buf** is an internal buffer.

3 If **s** is not a null pointer, the **wcrtomb** function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by **wc** (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by **s**. At most **MB_CUR_MAX** bytes are stored. If **wc** is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

4 The **wcrtomb** function returns the number of bytes stored in the array object (including any shift sequences). When **wc** is not a valid wide character, an encoding error occurs: the function stores the value of the macro **EILSEQ** in **errno** and returns **(size_t)(-1)**; the conversion state is unspecified.

7.29.6.4 Restartable multibyte/wide string conversion functions

1 These functions differ from the corresponding multibyte string functions of 7.22.8 (**mbstowcs** and **wcstombs**) in that they have an extra parameter, **ps**, of type pointer to **mbstate_t** that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If **ps** is a null pointer, each function uses its own internal **mbstate_t** object instead, which is initialized at program startup to the initial conversion state; the functions are not required to avoid data races with other calls to the same function in this case. The implementation behaves as if no library function calls these functions with a null pointer for **ps**.

2 Also unlike their corresponding functions, the conversion source parameter, **src**, has a pointer-to-pointer type. When the function is storing the results of conversions (that is, when **dst** is not a null pointer), the pointer object pointed to by this parameter is updated to reflect the amount of the source processed by that invocation.

7.29.6.4.1 The `mbsrtowcs` function

Synopsis

1
```
#include <wchar.h>
size_t mbsrtowcs(wchar_t * restrict dst,
     const char ** restrict src,
     size_t len,
     mbstate_t * restrict ps);
```

Description

2 The **`mbsrtowcs`** function converts a sequence of multibyte characters that begins in the conversion state described by the object pointed to by **`ps`**, from the array indirectly pointed to by **`src`** into a sequence of corresponding wide characters. If **`dst`** is not a null pointer, the converted characters are stored into the array pointed to by **`dst`**. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if **`dst`** is not a null pointer) when **`len`** wide characters have been stored into the array pointed to by **`dst`**.[350] Each conversion takes place as if by a call to the **`mbrtowc`** function.

3 If **`dst`** is not a null pointer, the pointer object pointed to by **`src`** is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multibyte character converted (if any). If conversion stopped due to reaching a terminating null character and if **`dst`** is not a null pointer, the resulting state described is the initial conversion state.

Returns

4 If the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the **`mbsrtowcs`** function stores the value of the macro **`EILSEQ`** in **`errno`** and returns **`(size_t)(-1)`**; the conversion state is unspecified. Otherwise, it returns the number of multibyte characters successfully converted, not including the terminating null character (if any).

350) Thus, the value of **`len`** is ignored if **`dst`** is a null pointer.

7.29.6.4.2 The `wcsrtombs` function

Synopsis

1
```
#include <wchar.h>
size_t wcsrtombs(char * restrict dst,
     const wchar_t ** restrict src,
     size_t len,
     mbstate_t * restrict ps);
```

Description

2 The **wcsrtombs** function converts a sequence of wide characters from the array indirectly pointed to by **src** into a sequence of corresponding multibyte characters that begins in the conversion state described by the object pointed to by **ps**. If **dst** is not a null pointer, the converted characters are then stored into the array pointed to by **dst**. Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier in two cases: when a wide character is reached that does not correspond to a valid multibyte character, or (if **dst** is not a null pointer) when the next multibyte character would exceed the limit of **len** total bytes to be stored into the array pointed to by **dst**. Each conversion takes place as if by a call to the **wcrtomb** function.[351)]

3 If **dst** is not a null pointer, the pointer object pointed to by **src** is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

Returns

4 If conversion stops because a wide character is reached that does not correspond to a valid multibyte character, an encoding error occurs: the **wcsrtombs** function stores the value of the macro **EILSEQ** in **errno** and returns **(size_t)(-1)**; the conversion state is unspecified. Otherwise, it returns the number of bytes in the resulting multibyte character sequence, not including the terminating null character (if any).

---

351) If conversion stops because a terminating null wide character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte.

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

7.30.1 Introduction

1 The header **`<wctype.h>`** defines one macro, and declares three data types and many functions.[352]

2 The types declared are

`wint_t`

described in 7.29.1;

`wctrans_t`

which is a scalar type that can hold values which represent locale-specific character mappings; and

`wctype_t`

which is a scalar type that can hold values which represent locale-specific character classifications.

3 The macro defined is **`WEOF`** (described in 7.29.1).

4 The functions declared are grouped as follows:

— Functions that provide wide character classification;

— Extensible functions that provide wide character classification;

— Functions that provide wide character case mapping;

— Extensible functions that provide wide character mapping.

5 For all functions described in this subclause that accept an argument of type **`wint_t`**, the value shall be representable as a **`wchar_t`** or shall equal the value of the macro **`WEOF`**. If this argument has any other value, the behavior is undefined.

6 The behavior of these functions is affected by the **`LC_CTYPE`** category of the current locale.

352) See ‘‘future library directions’’ (7.31.17).

7.30.2 Wide character classification utilities

1 The header **<wctype.h>** declares several functions useful for classifying wide characters.

2 The term *printing wide character* refers to a member of a locale-specific set of wide characters, each of which occupies at least one printing position on a display device. The term *control wide character* refers to a member of a locale-specific set of wide characters that are not printing wide characters.

7.30.2.1 Wide character classification functions

1 The functions in this subclause return nonzero (true) if and only if the value of the argument **wc** conforms to that in the description of the function.

2 Each of the following functions returns true for each wide character that corresponds (as if by a call to the **wctob** function) to a single-byte character for which the corresponding character classification function from 7.4.1 returns true, except that the **iswgraph** and **iswpunct** functions may differ with respect to wide characters other than **L' '** that are both printing and white-space wide characters.[353]

Forward references: the **wctob** function (7.29.6.1.2).

7.30.2.1.1 The iswalnum function

Synopsis

1
```
#include <wctype.h>
int iswalnum(wint_t wc);
```

Description

2 The **iswalnum** function tests for any wide character for which **iswalpha** or **iswdigit** is true.

7.30.2.1.2 The iswalpha function

Synopsis

1
```
#include <wctype.h>
int iswalpha(wint_t wc);
```

Description

2 The **iswalpha** function tests for any wide character for which **iswupper** or **iswlower** is true, or any wide character that is one of a locale-specific set of alphabetic

353) For example, if the expression **isalpha(wctob(wc))** evaluates to true, then the call **iswalpha(wc)** also returns true. But, if the expression **isgraph(wctob(wc))** evaluates to true (which cannot occur for **wc == L' '** of course), then either **iswgraph(wc)** or **iswprint(wc) && iswspace(wc)** is true, but not both.

wide characters for which none of **`iswcntrl`**, **`iswdigit`**, **`iswpunct`**, or **`iswspace`** is true.[354)]

7.30.2.1.3 The `iswblank` function

Synopsis

1
```
#include <wctype.h>
int iswblank(wint_t wc);
```

Description

2 The **`iswblank`** function tests for any wide character that is a standard blank wide character or is one of a locale-specific set of wide characters for which **`iswspace`** is true and that is used to separate words within a line of text. The standard blank wide characters are the following: space (**`L' '`**), and horizontal tab (**`L'\t'`**). In the **`"C"`** locale, **`iswblank`** returns true only for the standard blank characters.

7.30.2.1.4 The `iswcntrl` function

Synopsis

1
```
#include <wctype.h>
int iswcntrl(wint_t wc);
```

Description

2 The **`iswcntrl`** function tests for any control wide character.

7.30.2.1.5 The `iswdigit` function

Synopsis

1
```
#include <wctype.h>
int iswdigit(wint_t wc);
```

Description

2 The **`iswdigit`** function tests for any wide character that corresponds to a decimal-digit character (as defined in 5.2.1).

7.30.2.1.6 The `iswgraph` function

Synopsis

1
```
#include <wctype.h>
int iswgraph(wint_t wc);
```

354) The functions **`iswlower`** and **`iswupper`** test true or false separately for each of these additional wide characters; all four combinations are possible.

Description

2 The `iswgraph` function tests for any wide character for which `iswprint` is true and `iswspace` is false.[355]

7.30.2.1.7 The `iswlower` function

Synopsis

1
```
#include <wctype.h>
int iswlower(wint_t wc);
```

Description

2 The `iswlower` function tests for any wide character that corresponds to a lowercase letter or is one of a locale-specific set of wide characters for which none of `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` is true.

7.30.2.1.8 The `iswprint` function

Synopsis

1
```
#include <wctype.h>
int iswprint(wint_t wc);
```

Description

2 The `iswprint` function tests for any printing wide character.

7.30.2.1.9 The `iswpunct` function

Synopsis

1
```
#include <wctype.h>
int iswpunct(wint_t wc);
```

Description

2 The `iswpunct` function tests for any printing wide character that is one of a locale-specific set of punctuation wide characters for which neither `iswspace` nor `iswalnum` is true.[355]

7.30.2.1.10 The `iswspace` function

Synopsis

1
```
#include <wctype.h>
int iswspace(wint_t wc);
```

---

355) Note that the behavior of the `iswgraph` and `iswpunct` functions may differ from their corresponding functions in 7.4.1 with respect to printing, white-space, single-byte execution characters other than `' '`.

Description

2 The `iswspace` function tests for any wide character that corresponds to a locale-specific set of white-space wide characters for which none of `iswalnum`, `iswgraph`, or `iswpunct` is true.

7.30.2.1.11 The `iswupper` function

Synopsis

1
```
#include <wctype.h>
int iswupper(wint_t wc);
```

Description

2 The `iswupper` function tests for any wide character that corresponds to an uppercase letter or is one of a locale-specific set of wide characters for which none of `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` is true.

7.30.2.1.12 The `iswxdigit` function

Synopsis

1
```
#include <wctype.h>
int iswxdigit(wint_t wc);
```

Description

2 The `iswxdigit` function tests for any wide character that corresponds to a hexadecimal-digit character (as defined in 6.4.4.1).

7.30.2.2 Extensible wide character classification functions

1 The functions `wctype` and `iswctype` provide extensible wide character classification as well as testing equivalent to that performed by the functions described in the previous subclause (7.30.2.1).

7.30.2.2.1 The `iswctype` function

Synopsis

1
```
#include <wctype.h>
int iswctype(wint_t wc, wctype_t desc);
```

Description

2 The `iswctype` function determines whether the wide character `wc` has the property described by `desc`. The current setting of the `LC_CTYPE` category shall be the same as during the call to `wctype` that returned the value `desc`.

3 Each of the following expressions has a truth-value equivalent to the call to the wide character classification function (7.30.2.1) in the comment that follows the expression:

```
iswctype(wc, wctype("alnum"))     // iswalnum(wc)
iswctype(wc, wctype("alpha"))     // iswalpha(wc)
iswctype(wc, wctype("blank"))     // iswblank(wc)
iswctype(wc, wctype("cntrl"))     // iswcntrl(wc)
iswctype(wc, wctype("digit"))     // iswdigit(wc)
iswctype(wc, wctype("graph"))     // iswgraph(wc)
iswctype(wc, wctype("lower"))     // iswlower(wc)
iswctype(wc, wctype("print"))     // iswprint(wc)
iswctype(wc, wctype("punct"))     // iswpunct(wc)
iswctype(wc, wctype("space"))     // iswspace(wc)
iswctype(wc, wctype("upper"))     // iswupper(wc)
iswctype(wc, wctype("xdigit"))    // iswxdigit(wc)
```

Returns

4 The **iswctype** function returns nonzero (true) if and only if the value of the wide character **wc** has the property described by **desc**. If **desc** is zero, the **iswctype** function returns zero (false).

Forward references: the **wctype** function (7.30.2.2.2).

7.30.2.2.2 The wctype function

Synopsis

1
```
#include <wctype.h>
wctype_t wctype(const char *property);
```

Description

2 The **wctype** function constructs a value with type **wctype_t** that describes a class of wide characters identified by the string argument **property**.

3 The strings listed in the description of the **iswctype** function shall be valid in all locales as **property** arguments to the **wctype** function.

Returns

4 If **property** identifies a valid class of wide characters according to the **LC_CTYPE** category of the current locale, the **wctype** function returns a nonzero value that is valid as the second argument to the **iswctype** function; otherwise, it returns zero.

7.30.3 Wide character case mapping utilities

1 The header `<wctype.h>` declares several functions useful for mapping wide characters.

7.30.3.1 Wide character case mapping functions

7.30.3.1.1 The `towlower` function

Synopsis

1
```
#include <wctype.h>
wint_t towlower(wint_t wc);
```

Description

2 The **towlower** function converts an uppercase letter to a corresponding lowercase letter.

Returns

3 If the argument is a wide character for which **iswupper** is true and there are one or more corresponding wide characters, as specified by the current locale, for which **iswlower** is true, the **towlower** function returns one of the corresponding wide characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.30.3.1.2 The `towupper` function

Synopsis

1
```
#include <wctype.h>
wint_t towupper(wint_t wc);
```

Description

2 The **towupper** function converts a lowercase letter to a corresponding uppercase letter.

Returns

3 If the argument is a wide character for which **iswlower** is true and there are one or more corresponding wide characters, as specified by the current locale, for which **iswupper** is true, the **towupper** function returns one of the corresponding wide characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.30.3.2 Extensible wide character case mapping functions

1 The functions **wctrans** and **towctrans** provide extensible wide character mapping as well as case mapping equivalent to that performed by the functions described in the previous subclause (7.30.3.1).

7.30.3.2.1 The `towctrans` function

Synopsis

1
```
#include <wctype.h>
wint_t towctrans(wint_t wc, wctrans_t desc);
```

Description

2 The `towctrans` function maps the wide character `wc` using the mapping described by `desc`. The current setting of the `LC_CTYPE` category shall be the same as during the call to `wctrans` that returned the value `desc`.

3 Each of the following expressions behaves the same as the call to the wide character case mapping function (7.30.3.1) in the comment that follows the expression:

```
towctrans(wc, wctrans("tolower"))     // towlower(wc)
towctrans(wc, wctrans("toupper"))     // towupper(wc)
```

Returns

4 The `towctrans` function returns the mapped value of `wc` using the mapping described by `desc`. If `desc` is zero, the `towctrans` function returns the value of `wc`.

7.30.3.2.2 The `wctrans` function

Synopsis

1
```
#include <wctype.h>
wctrans_t wctrans(const char *property);
```

Description

2 The `wctrans` function constructs a value with type `wctrans_t` that describes a mapping between wide characters identified by the string argument `property`.

3 The strings listed in the description of the `towctrans` function shall be valid in all locales as `property` arguments to the `wctrans` function.

Returns

4 If `property` identifies a valid mapping of wide characters according to the `LC_CTYPE` category of the current locale, the `wctrans` function returns a nonzero value that is valid as the second argument to the `towctrans` function; otherwise, it returns zero.

7.31 Future library directions

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

7.31.1 Complex arithmetic `<complex.h>`

1 The function names

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

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

7.31.2 Character handling `<ctype.h>`

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

7.31.3 Errors `<errno.h>`

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

7.31.4 Floating-point environment `<fenv.h>`

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

7.31.5 Format conversion of integer types `<inttypes.h>`

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

7.31.6 Localization `<locale.h>`

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

7.31.7 Signal handling `<signal.h>`

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

7.31.8 Atomics `<stdatomic.h>`

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

and a lowercase letter may be added to the definition of the **memory_order** type in the **<stdatomic.h>** header. Function names that begin with **atomic_** and a lowercase letter may be added to the declarations in the **<stdatomic.h>** header.

7.31.9 Boolean type and values <stdbool.h>

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

7.31.10 Integer types <stdint.h>

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

7.31.11 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.

7.31.12 General utilities <stdlib.h>

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

7.31.13 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.

7.31.14 Date and time <time.h>

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

7.31.15 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.

7.31.16 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.

7.31.17 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.

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

```
auto          * if          unsigned
break           inline      void
case            int         volatile
char            long        while
const           register    _Alignas
continue        restrict    _Alignof
default         return      _Atomic
do              short       _Bool
double          signed      _Complex
else            sizeof      _Generic
enum            static      _Imaginary
extern          struct      _Noreturn
float           switch      _Static_assert
for             typedef     _Thread_local
goto            union
```

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

(6.4.4.1) *integer-constant:*
    decimal-constant integer-suffix$_{opt}$
    octal-constant integer-suffix$_{opt}$
    hexadecimal-constant integer-suffix$_{opt}$

(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
```

(6.4.4.1) *integer-suffix:*
unsigned-suffix *long-suffix*$_{opt}$
unsigned-suffix *long-long-suffix*
long-suffix *unsigned-suffix*$_{opt}$
long-long-suffix *unsigned-suffix*$_{opt}$

(6.4.4.1) *unsigned-suffix:* one of
u U

(6.4.4.1) *long-suffix:* one of
l L

(6.4.4.1) *long-long-suffix:* one of
ll LL

(6.4.4.2) *floating-constant:*
decimal-floating-constant
hexadecimal-floating-constant

(6.4.4.2) *decimal-floating-constant:*
fractional-constant *exponent-part*$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence *exponent-part* *floating-suffix*$_{opt}$

(6.4.4.2) *hexadecimal-floating-constant:*
hexadecimal-prefix *hexadecimal-fractional-constant*
binary-exponent-part *floating-suffix*$_{opt}$
hexadecimal-prefix *hexadecimal-digit-sequence*
binary-exponent-part *floating-suffix*$_{opt}$

(6.4.4.2) *fractional-constant:*
digit-sequence$_{opt}$ **.** *digit-sequence*
digit-sequence **.**

(6.4.4.2) *exponent-part:*
e *sign*$_{opt}$ *digit-sequence*
E *sign*$_{opt}$ *digit-sequence*

(6.4.4.2) *sign:* one of
+ -

(6.4.4.2) *digit-sequence:*
digit
digit-sequence digit

(6.4.4.2) *hexadecimal-fractional-constant:*
hexadecimal-digit-sequence$_{opt}$ **.**
hexadecimal-digit-sequence
hexadecimal-digit-sequence **.**

(6.4.4.2) *binary-exponent-part:*
p *sign*$_{opt}$ *digit-sequence*
P *sign*$_{opt}$ *digit-sequence*

(6.4.4.2) *hexadecimal-digit-sequence:*
hexadecimal-digit
hexadecimal-digit-sequence hexadecimal-digit

(6.4.4.2) *floating-suffix:* one of
f l F L

(6.4.4.3) *enumeration-constant:*
identifier

(6.4.4.4) *character-constant:*
' *c-char-sequence* **'**
L' *c-char-sequence* **'**
u' *c-char-sequence* **'**
U' *c-char-sequence* **'**

(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.1.6 String literals

(6.4.5) *string-literal:*

encoding-prefix$_{opt}$ **"** *s-char-sequence*$_{opt}$ **"**

(6.4.5) *encoding-prefix:*

```
u8
u
U
L
```

(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

A.1.7 Punctuators

(6.4.6) *punctuator:* one of

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

A.1.8 Header names

(6.4.7) *header-name:*
  < *h-char-sequence* **>**
  " *q-char-sequence* **"**

(6.4.7) *h-char-sequence:*
  h-char
  h-char-sequence h-char

(6.4.7) *h-char:*
  any member of the source character set except
    the new-line character and **>**

(6.4.7) *q-char-sequence:*
  q-char
  q-char-sequence q-char

(6.4.7) *q-char:*
  any member of the source character set except
    the new-line character and **"**

A.1.9 Preprocessing numbers

(6.4.8) *pp-number:*
  digit
  . *digit*
  pp-number digit
  pp-number identifier-nondigit
  pp-number **e** *sign*
  pp-number **E** *sign*
  pp-number **p** *sign*
  pp-number **P** *sign*
  pp-number **.**

A.2 Phrase structure grammar

A.2.1 Expressions

(6.5.1) *primary-expression:*
identifier
constant
string-literal
(*expression* **)**
generic-selection

(6.5.1.1) *generic-selection:*
_Generic (*assignment-expression* **,** *generic-assoc-list* **)**

(6.5.1.1) *generic-assoc-list:*
generic-association
generic-assoc-list **,** *generic-association*

(6.5.1.1) *generic-association:*
type-name **:** *assignment-expression*
default : *assignment-expression*

(6.5.2) *postfix-expression:*
primary-expression
postfix-expression **[** *expression* **]**
postfix-expression **(** $argument\text{-}expression\text{-}list_{opt}$ **)**
postfix-expression **.** *identifier*
postfix-expression **->** *identifier*
postfix-expression **++**
postfix-expression **--**
(*type-name* **) {** *initializer-list* **}**
(*type-name* **) {** *initializer-list* **, }**

(6.5.2) *argument-expression-list:*
assignment-expression
argument-expression-list **,** *assignment-expression*

(6.5.3) *unary-expression:*
postfix-expression
++ *unary-expression*
-- *unary-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof (*type-name* **)**
_Alignof (*type-name* **)**

(6.5.3) *unary-operator:* one of
`& * + - ~ !`

(6.5.4) *cast-expression:*
unary-expression
`(` *type-name* `)` *cast-expression*

(6.5.5) *multiplicative-expression:*
cast-expression
multiplicative-expression `*` *cast-expression*
multiplicative-expression `/` *cast-expression*
multiplicative-expression `%` *cast-expression*

(6.5.6) *additive-expression:*
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

(6.5.7) *shift-expression:*
additive-expression
shift-expression `<<` *additive-expression*
shift-expression `>>` *additive-expression*

(6.5.8) *relational-expression:*
shift-expression
relational-expression `<` *shift-expression*
relational-expression `>` *shift-expression*
relational-expression `<=` *shift-expression*
relational-expression `>=` *shift-expression*

(6.5.9) *equality-expression:*
relational-expression
equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

(6.5.10) *AND-expression:*
equality-expression
AND-expression `&` *equality-expression*

(6.5.11) *exclusive-OR-expression:*
AND-expression
exclusive-OR-expression `^` *AND-expression*

(6.5.12) *inclusive-OR-expression:*
exclusive-OR-expression
inclusive-OR-expression **|** *exclusive-OR-expression*

(6.5.13) *logical-AND-expression:*
inclusive-OR-expression
logical-AND-expression **&&** *inclusive-OR-expression*

(6.5.14) *logical-OR-expression:*
logical-AND-expression
logical-OR-expression **||** *logical-AND-expression*

(6.5.15) *conditional-expression:*
logical-OR-expression
logical-OR-expression **?** *expression* **:** *conditional-expression*

(6.5.16) *assignment-expression:*
conditional-expression
unary-expression *assignment-operator* *assignment-expression*

(6.5.16) *assignment-operator:* one of
= *= /= %= += -= <<= >>= &= ^= |=

(6.5.17) *expression:*
assignment-expression
expression **,** *assignment-expression*

(6.6) *constant-expression:*
conditional-expression

A.2.2 Declarations

(6.7) *declaration:*
declaration-specifiers *init-declarator-list*$_{opt}$ **;**
static_assert-declaration

(6.7) *declaration-specifiers:*
storage-class-specifier *declaration-specifiers*$_{opt}$
type-specifier *declaration-specifiers*$_{opt}$
type-qualifier *declaration-specifiers*$_{opt}$
function-specifier *declaration-specifiers*$_{opt}$
alignment-specifier *declaration-specifiers*$_{opt}$

(6.7) *init-declarator-list:*
init-declarator
init-declarator-list **,** *init-declarator*

(6.7) *init-declarator:*
declarator
declarator **=** *initializer*

(6.7.1) *storage-class-specifier:*
typedef
extern
static
_Thread_local
auto
register

(6.7.2) *type-specifier:*
void
char
short
int
long
float
double
signed
unsigned
_Bool
_Complex
atomic-type-specifier
struct-or-union-specifier
enum-specifier
typedef-name

(6.7.2.1) *struct-or-union-specifier:*
struct-or-union identifier$_{opt}$ **{** *struct-declaration-list* **}**
struct-or-union identifier

(6.7.2.1) *struct-or-union:*
struct
union

(6.7.2.1) *struct-declaration-list:*
struct-declaration
struct-declaration-list struct-declaration

(6.7.2.1) *struct-declaration:*
specifier-qualifier-list struct-declarator-list$_{opt}$ **;**
static_assert-declaration

(6.7.2.1) *specifier-qualifier-list:*
type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

(6.7.2.1) *struct-declarator-list:*
struct-declarator
struct-declarator-list **,** *struct-declarator*

(6.7.2.1) *struct-declarator:*
declarator
declarator$_{opt}$ **:** *constant-expression*

(6.7.2.2) *enum-specifier:*
enum *identifier*$_{opt}$ **{** *enumerator-list* **}**
enum *identifier*$_{opt}$ **{** *enumerator-list* **, }**
enum *identifier*

(6.7.2.2) *enumerator-list:*
enumerator
enumerator-list **,** *enumerator*

(6.7.2.2) *enumerator:*
enumeration-constant
enumeration-constant **=** *constant-expression*

(6.7.2.4) *atomic-type-specifier:*
_Atomic (*type-name* **)**

(6.7.3) *type-qualifier:*
const
restrict
volatile
_Atomic

(6.7.4) *function-specifier:*
inline
_Noreturn

(6.7.5) *alignment-specifier:*
_Alignas (*type-name* **)**
_Alignas (*constant-expression* **)**

(6.7.6) *declarator:*
pointer$_{opt}$ *direct-declarator*

(6.7.6) *direct-declarator:*
identifier
(*declarator* **)**
direct-declarator **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
direct-declarator **[`static`** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list* **`static`** *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list*$_{opt}$ ***]**
direct-declarator **(** *parameter-type-list* **)**
direct-declarator **(** *identifier-list*$_{opt}$ **)**

(6.7.6) *pointer:*
***** *type-qualifier-list*$_{opt}$
***** *type-qualifier-list*$_{opt}$ *pointer*

(6.7.6) *type-qualifier-list:*
type-qualifier
type-qualifier-list *type-qualifier*

(6.7.6) *parameter-type-list:*
parameter-list
parameter-list **, ...**

(6.7.6) *parameter-list:*
parameter-declaration
parameter-list **,** *parameter-declaration*

(6.7.6) *parameter-declaration:*
declaration-specifiers *declarator*
declaration-specifiers *abstract-declarator*$_{opt}$

(6.7.6) *identifier-list:*
identifier
identifier-list **,** *identifier*

(6.7.7) *type-name:*
specifier-qualifier-list *abstract-declarator*$_{opt}$

(6.7.7) *abstract-declarator:*
pointer
pointer$_{opt}$ *direct-abstract-declarator*

(6.7.7) *direct-abstract-declarator:*
   (*abstract-declarator* **)**
   direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list*$_{opt}$
      assignment-expression$_{opt}$ **]**
   direct-abstract-declarator$_{opt}$ **[static** *type-qualifier-list*$_{opt}$
      assignment-expression **]**
   direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list* **static**
      assignment-expression **]**
   direct-abstract-declarator$_{opt}$ **[*]**
   direct-abstract-declarator$_{opt}$ **(** *parameter-type-list*$_{opt}$ **)**

(6.7.8) *typedef-name:*
   identifier

(6.7.9) *initializer:*
   assignment-expression
   { *initializer-list* **}**
   { *initializer-list* **, }**

(6.7.9) *initializer-list:*
   designation$_{opt}$ *initializer*
   initializer-list **,** *designation*$_{opt}$ *initializer*

(6.7.9) *designation:*
   designator-list **=**

(6.7.9) *designator-list:*
   designator
   designator-list designator

(6.7.9) *designator:*
   [*constant-expression* **]**
   . *identifier*

(6.7.10) *static_assert-declaration:*
   _Static_assert (*constant-expression* **,** *string-literal* **) ;**

A.2.3 Statements

(6.8) *statement:*
labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement

(6.8.1) *labeled-statement:*
identifier **:** *statement*
case *constant-expression* **:** *statement*
default : *statement*

(6.8.2) *compound-statement:*
{ *block-item-list*$_{opt}$ **}**

(6.8.2) *block-item-list:*
block-item
block-item-list block-item

(6.8.2) *block-item:*
declaration
statement

(6.8.3) *expression-statement:*
expression$_{opt}$ **;**

(6.8.4) *selection-statement:*
if (*expression* **)** *statement*
if (*expression* **)** *statement* **else** *statement*
switch (*expression* **)** *statement*

(6.8.5) *iteration-statement:*
while (*expression* **)** *statement*
do *statement* **while (** *expression* **) ;**
for (*expression*$_{opt}$ **;** *expression*$_{opt}$ **;** *expression*$_{opt}$ **)** *statement*
for (*declaration expression*$_{opt}$ **;** *expression*$_{opt}$ **)** *statement*

(6.8.6) *jump-statement:*
goto *identifier* **;**
continue ;
break ;
return *expression*$_{opt}$ **;**

A.2.4 External definitions

(6.9) *translation-unit:*
    external-declaration
    translation-unit external-declaration

(6.9) *external-declaration:*
    function-definition
    declaration

(6.9.1) *function-definition:*
    declaration-specifiers declarator $declaration\text{-}list_{opt}$ *compound-statement*

(6.9.1) *declaration-list:*
    declaration
    declaration-list declaration

A.3 Preprocessing directives

(6.10) *preprocessing-file:*
    $group_{opt}$

(6.10) *group:*
    group-part
    group group-part

(6.10) *group-part:*
    if-section
    control-line
    text-line
    # *non-directive*

(6.10) *if-section:*
    if-group $elif\text{-}groups_{opt}$ $else\text{-}group_{opt}$ *endif-line*

(6.10) *if-group:*
    # if *constant-expression new-line* $group_{opt}$
    # ifdef *identifier new-line* $group_{opt}$
    # ifndef *identifier new-line* $group_{opt}$

(6.10) *elif-groups:*
    elif-group
    elif-groups elif-group

(6.10) *elif-group:*
    # elif *constant-expression new-line* $group_{opt}$

(6.10) *else-group:*
  # else *new-line group*$_{opt}$

(6.10) *endif-line:*
  # endif *new-line*

(6.10) *control-line:*
  # include *pp-tokens new-line*
  # define *identifier replacement-list new-line*
  # define *identifier lparen identifier-list*$_{opt}$ **)**
    replacement-list new-line
  # define *identifier lparen* **...)** *replacement-list new-line*
  # define *identifier lparen identifier-list* **, ...)**
    replacement-list new-line
  # undef *identifier new-line*
  # line *pp-tokens new-line*
  # error *pp-tokens*$_{opt}$ *new-line*
  # pragma *pp-tokens*$_{opt}$ *new-line*
  # *new-line*

(6.10) *text-line:*
  pp-tokens$_{opt}$ *new-line*

(6.10) *non-directive:*
  pp-tokens new-line

(6.10) *lparen:*
  a **(** character not immediately preceded by white-space

(6.10) *replacement-list:*
  pp-tokens$_{opt}$

(6.10) *pp-tokens:*
  preprocessing-token
  pp-tokens preprocessing-token

(6.10) *new-line:*
  the new-line character

Annex B

(informative)

Library summary

B.1 Diagnostics <assert.h>

```
NDEBUG
static_assert
```

`void assert(`*scalar* `expression);`

B.2 Complex <complex.h>

```
__STDC_NO_COMPLEX__        imaginary
complex                    _Imaginary_I
_Complex_I                 I
```

`#pragma STDC CX_LIMITED_RANGE` *on-off-switch*

```
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

```
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x,
     long double complex y);
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
double complex CMPLX(double x, double y);
float complex CMPLXF(float x, float y);
long double complex CMPLXL(long double x, long double y);
double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
double complex cproj(double complex z);
```

```
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

B.3 Character handling `<ctype.h>`

```
int isalnum(int c);
int isalpha(int c);
int isblank(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int tolower(int c);
int toupper(int c);
```

B.4 Errors `<errno.h>`

```
EDOM           EILSEQ         ERANGE         errno

__STDC_WANT_LIB_EXT1__
errno_t
```

B.5 Floating-point environment `<fenv.h>`

```
fenv_t              FE_OVERFLOW         FE_TOWARDZERO
fexcept_t           FE_UNDERFLOW        FE_UPWARD
FE_DIVBYZERO        FE_ALL_EXCEPT       FE_DFL_ENV
FE_INEXACT          FE_DOWNWARD
FE_INVALID          FE_TONEAREST
```

```
#pragma STDC FENV_ACCESS on-off-switch
int feclearexcept(int excepts);
int fegetexceptflag(fexcept_t *flagp, int excepts);
int feraiseexcept(int excepts);
int fesetexceptflag(const fexcept_t *flagp,
     int excepts);
int fetestexcept(int excepts);
```

```
int fegetround(void);
int fesetround(int round);
int fegetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
int fesetenv(const fenv_t *envp);
int feupdateenv(const fenv_t *envp);
```

B.6 Characteristics of floating types `<float.h>`

```
FLT_ROUNDS           DBL_DIG              FLT_MAX
FLT_EVAL_METHOD      LDBL_DIG             DBL_MAX
FLT_HAS_SUBNORM      FLT_MIN_EXP          LDBL_MAX
DBL_HAS_SUBNORM      DBL_MIN_EXP          FLT_EPSILON
LDBL_HAS_SUBNORM     LDBL_MIN_EXP         DBL_EPSILON
FLT_RADIX            FLT_MIN_10_EXP       LDBL_EPSILON
FLT_MANT_DIG         DBL_MIN_10_EXP       FLT_MIN
DBL_MANT_DIG         LDBL_MIN_10_EXP      DBL_MIN
LDBL_MANT_DIG        FLT_MAX_EXP          LDBL_MIN
FLT_DECIMAL_DIG      DBL_MAX_EXP          FLT_TRUE_MIN
DBL_DECIMAL_DIG      LDBL_MAX_EXP         DBL_TRUE_MIN
LDBL_DECIMAL_DIG     FLT_MAX_10_EXP       LDBL_TRUE_MIN
DECIMAL_DIG          DBL_MAX_10_EXP
FLT_DIG              LDBL_MAX_10_EXP
```

B.7 Format conversion of integer types `<inttypes.h>`

```
imaxdiv_t
```

```
PRIdN       PRIdLEASTN      PRIdFASTN      PRIdMAX      PRIdPTR
PRIiN       PRIiLEASTN      PRIiFASTN      PRIiMAX      PRIiPTR
PRIoN       PRIoLEASTN      PRIoFASTN      PRIoMAX      PRIoPTR
PRIuN       PRIuLEASTN      PRIuFASTN      PRIuMAX      PRIuPTR
PRIxN       PRIxLEASTN      PRIxFASTN      PRIxMAX      PRIxPTR
PRIXN       PRIXLEASTN      PRIXFASTN      PRIXMAX      PRIXPTR
SCNdN       SCNdLEASTN      SCNdFASTN      SCNdMAX      SCNdPTR
SCNiN       SCNiLEASTN      SCNiFASTN      SCNiMAX      SCNiPTR
SCNoN       SCNoLEASTN      SCNoFASTN      SCNoMAX      SCNoPTR
SCNuN       SCNuLEASTN      SCNuFASTN      SCNuMAX      SCNuPTR
SCNxN       SCNxLEASTN      SCNxFASTN      SCNxMAX      SCNxPTR
```

```
intmax_t imaxabs(intmax_t j);
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
intmax_t strtoimax(const char * restrict nptr,
        char ** restrict endptr, int base);
```

```
uintmax_t strtoumax(const char * restrict nptr,
        char ** restrict endptr, int base);
intmax_t wcstoimax(const wchar_t * restrict nptr,
        wchar_t ** restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t * restrict nptr,
        wchar_t ** restrict endptr, int base);
```

B.8 Alternative spellings `<iso646.h>`

```
and           bitor         not_eq        xor
and_eq        compl         or            xor_eq
bitand        not           or_eq
```

B.9 Sizes of integer types `<limits.h>`

```
CHAR_BIT      CHAR_MAX      INT_MIN       ULONG_MAX
SCHAR_MIN     MB_LEN_MAX    INT_MAX       LLONG_MIN
SCHAR_MAX     SHRT_MIN      UINT_MAX      LLONG_MAX
UCHAR_MAX     SHRT_MAX      LONG_MIN      ULLONG_MAX
CHAR_MIN      USHRT_MAX     LONG_MAX
```

B.10 Localization `<locale.h>`

```
struct lconv  LC_ALL        LC_CTYPE      LC_NUMERIC
NULL          LC_COLLATE    LC_MONETARY   LC_TIME
```

```
char *setlocale(int category, const char *locale);
struct lconv *localeconv(void);
```

B.11 Mathematics `<math.h>`

```
float_t          FP_INFINITE      FP_FAST_FMAL
double_t         FP_NAN           FP_ILOGB0
HUGE_VAL         FP_NORMAL        FP_ILOGBNAN
HUGE_VALF        FP_SUBNORMAL     MATH_ERRNO
HUGE_VALL        FP_ZERO          MATH_ERREXCEPT
INFINITY         FP_FAST_FMA      math_errhandling
NAN              FP_FAST_FMAF
```

```
#pragma STDC FP_CONTRACT on-off-switch
int fpclassify(real-floating x);
int isfinite(real-floating x);
int isinf(real-floating x);
int isnan(real-floating x);
int isnormal(real-floating x);
int signbit(real-floating x);
```

```
double acos(double x);
float acosf(float x);
long double acosl(long double x);
double asin(double x);
float asinf(float x);
long double asinl(long double x);
double atan(double x);
float atanf(float x);
long double atanl(long double x);
double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
double cos(double x);
float cosf(float x);
long double cosl(long double x);
double sin(double x);
float sinf(float x);
long double sinl(long double x);
double tan(double x);
float tanf(float x);
long double tanl(long double x);
double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
double cosh(double x);
float coshf(float x);
long double coshl(long double x);
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
double exp(double x);
float expf(float x);
```

```
long double expl(long double x);
double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
double frexp(double value, int *exp);
float frexpf(float value, int *exp);
long double frexpl(long double value, int *exp);
int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
double ldexp(double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
double log(double x);
float logf(float x);
long double logl(long double x);
double log10(double x);
float log10f(float x);
long double log10l(long double x);
double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
double log2(double x);
float log2f(float x);
long double log2l(long double x);
double logb(double x);
float logbf(float x);
long double logbl(long double x);
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
double scalbln(double x, long int n);
float scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);
double cbrt(double x);
```

```
float cbrtf(float x);
long double cbrtl(long double x);
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
double erf(double x);
float erff(float x);
long double erfl(long double x);
double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);
double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
double floor(double x);
float floorf(float x);
long double floorl(long double x);
double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
double rint(double x);
float rintf(float x);
long double rintl(long double x);
long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);
```

```
long long int llrint(double x);
long long int llrintf(float x);
long long int llrintl(long double x);
double round(double x);
float roundf(float x);
long double roundl(long double x);
long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
double trunc(double x);
float truncf(float x);
long double truncl(long double x);
double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y,
     int *quo);
double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
double fmax(double x, double y);
```

```
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y,
     long double z);
int isgreater(real-floating x, real-floating y);
int isgreaterequal(real-floating x, real-floating y);
int isless(real-floating x, real-floating y);
int islessequal(real-floating x, real-floating y);
int islessgreater(real-floating x, real-floating y);
int isunordered(real-floating x, real-floating y);
```

B.12 Nonlocal jumps `<setjmp.h>`

```
jmp_buf
```

```
int setjmp(jmp_buf env);
_Noreturn void longjmp(jmp_buf env, int val);
```

B.13 Signal handling `<signal.h>`

```
sig_atomic_t  SIG_IGN       SIGILL        SIGTERM
SIG_DFL       SIGABRT       SIGINT
SIG_ERR       SIGFPE        SIGSEGV
```

```
void (*signal(int sig, void (*func)(int)))(int);
int raise(int sig);
```

B.14 Alignment <stdalign.h>

```
alignas
__alignas_is_defined
```

B.15 Variable arguments <stdarg.h>

```
va_list
```

```
type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, parmN);
```

B.16 Atomics <stdatomic.h>

```
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
ATOMIC_FLAG_INIT
memory_order
atomic_flag
memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_acq_rel
memory_order_seq_cst
atomic_bool
atomic_char
atomic_schar
atomic_uchar
atomic_short
atomic_ushort
atomic_int
atomic_uint
atomic_long
atomic_ulong
atomic_llong
atomic_ullong
atomic_char16_t
atomic_char32_t
atomic_wchar_t
atomic_int_least8_t
atomic_uint_least8_t
atomic_int_least16_t
atomic_uint_least16_t
atomic_int_least32_t
atomic_uint_least32_t
atomic_int_least64_t
atomic_uint_least64_t
atomic_int_fast8_t
atomic_uint_fast8_t
atomic_int_fast16_t
atomic_uint_fast16_t
atomic_int_fast32_t
atomic_uint_fast32_t
atomic_int_fast64_t
atomic_uint_fast64_t
atomic_intptr_t
atomic_uintptr_t
```

```
atomic_size_t                atomic_intmax_t
atomic_ptrdiff_t             atomic_uintmax_t
#define ATOMIC_VAR_INIT(C value)
void atomic_init(volatile A *obj, C value);
type kill_dependency(type y);
void atomic_thread_fence(memory_order order);
void atomic_signal_fence(memory_order order);
_Bool atomic_is_lock_free(const volatile A *obj);
void atomic_store(volatile A *object, C desired);
void atomic_store_explicit(volatile A *object,
     C desired, memory_order order);
C atomic_load(volatile A *object);
C atomic_load_explicit(volatile A *object,
     memory_order order);
C atomic_exchange(volatile A *object, C desired);
C atomic_exchange_explicit(volatile A *object,
     C desired, memory_order order);
_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);
C atomic_fetch_key(volatile A *object, M operand);
C atomic_fetch_key_explicit(volatile A *object,
     M operand, memory_order order);
_Bool atomic_flag_test_and_set(
     volatile atomic_flag *object);
_Bool atomic_flag_test_and_set_explicit(
     volatile atomic_flag *object, memory_order order);
void atomic_flag_clear(volatile atomic_flag *object);
void atomic_flag_clear_explicit(
     volatile atomic_flag *object, memory_order order);
```

B.17 Boolean type and values <stdbool.h>

```
bool
true
false
__bool_true_false_are_defined
```

B.18 Common definitions <stddef.h>

```
ptrdiff_t     max_align_t   NULL
size_t        wchar_t
```

```
offsetof(type, member-designator)
```

```
__STDC_WANT_LIB_EXT1__
rsize_t
```

B.19 Integer types <stdint.h>

```
intN_t              INT_LEASTN_MIN      PTRDIFF_MAX
uintN_t             INT_LEASTN_MAX      SIG_ATOMIC_MIN
int_leastN_t        UINT_LEASTN_MAX     SIG_ATOMIC_MAX
uint_leastN_t       INT_FASTN_MIN       SIZE_MAX
int_fastN_t         INT_FASTN_MAX       WCHAR_MIN
uint_fastN_t        UINT_FASTN_MAX      WCHAR_MAX
intptr_t            INTPTR_MIN          WINT_MIN
uintptr_t           INTPTR_MAX          WINT_MAX
intmax_t            UINTPTR_MAX         INTN_C(value)
uintmax_t           INTMAX_MIN          UINTN_C(value)
INTN_MIN            INTMAX_MAX          INTMAX_C(value)
INTN_MAX            UINTMAX_MAX         UINTMAX_C(value)
UINTN_MAX           PTRDIFF_MIN
```

```
__STDC_WANT_LIB_EXT1__
RSIZE_MAX
```

B.20 Input/output `<stdio.h>`

```
size_t        _IOLBF        FILENAME_MAX  TMP_MAX
FILE          _IONBF        L_tmpnam      stderr
fpos_t        BUFSIZ        SEEK_CUR      stdin
NULL          EOF           SEEK_END      stdout
_IOFBF        FOPEN_MAX     SEEK_SET
```

```
int remove(const char *filename);
int rename(const char *old, const char *new);
FILE *tmpfile(void);
char *tmpnam(char *s);
int fclose(FILE *stream);
int fflush(FILE *stream);
FILE *fopen(const char * restrict filename,
     const char * restrict mode);
FILE *freopen(const char * restrict filename,
     const char * restrict mode,
     FILE * restrict stream);
void setbuf(FILE * restrict stream,
     char * restrict buf);
int setvbuf(FILE * restrict stream,
     char * restrict buf,
     int mode, size_t size);
int fprintf(FILE * restrict stream,
     const char * restrict format, ...);
int fscanf(FILE * restrict stream,
     const char * restrict format, ...);
int printf(const char * restrict format, ...);
int scanf(const char * restrict format, ...);
int snprintf(char * restrict s, size_t n,
     const char * restrict format, ...);
int sprintf(char * restrict s,
     const char * restrict format, ...);
int sscanf(const char * restrict s,
     const char * restrict format, ...);
int vfprintf(FILE * restrict stream,
     const char * restrict format, va_list arg);
int vfscanf(FILE * restrict stream,
     const char * restrict format, va_list arg);
int vprintf(const char * restrict format, va_list arg);
int vscanf(const char * restrict format, va_list arg);
```

```
int vsnprintf(char * restrict s, size_t n,
     const char * restrict format, va_list arg);
int vsprintf(char * restrict s,
     const char * restrict format, va_list arg);
int vsscanf(const char * restrict s,
     const char * restrict format, va_list arg);
int fgetc(FILE *stream);
char *fgets(char * restrict s, int n,
     FILE * restrict stream);
int fputc(int c, FILE *stream);
int fputs(const char * restrict s,
     FILE * restrict stream);
int getc(FILE *stream);
int getchar(void);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(const char *s);
int ungetc(int c, FILE *stream);
size_t fread(void * restrict ptr,
     size_t size, size_t nmemb,
     FILE * restrict stream);
size_t fwrite(const void * restrict ptr,
     size_t size, size_t nmemb,
     FILE * restrict stream);
int fgetpos(FILE * restrict stream,
     fpos_t * restrict pos);
int fseek(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos);
long int ftell(FILE *stream);
void rewind(FILE *stream);
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
void perror(const char *s);
```

```
__STDC_WANT_LIB_EXT1__
L_tmpnam_s    TMP_MAX_S     errno_t       rsize_t
```

```
errno_t tmpfile_s(FILE * restrict * restrict streamptr);
errno_t tmpnam_s(char *s, rsize_t maxsize);
```

```
errno_t fopen_s(FILE * restrict * restrict streamptr,
     const char * restrict filename,
     const char * restrict mode);
errno_t freopen_s(FILE * restrict * restrict newstreamptr,
     const char * restrict filename,
     const char * restrict mode,
     FILE * restrict stream);
int fprintf_s(FILE * restrict stream,
     const char * restrict format, ...);
int fscanf_s(FILE * restrict stream,
     const char * restrict format, ...);
int printf_s(const char * restrict format, ...);
int scanf_s(const char * restrict format, ...);
int snprintf_s(char * restrict s, rsize_t n,
     const char * restrict format, ...);
int sprintf_s(char * restrict s, rsize_t n,
     const char * restrict format, ...);
int sscanf_s(const char * restrict s,
     const char * restrict format, ...);
int vfprintf_s(FILE * restrict stream,
     const char * restrict format,
     va_list arg);
int vfscanf_s(FILE * restrict stream,
     const char * restrict format,
     va_list arg);
int vprintf_s(const char * restrict format,
     va_list arg);
int vscanf_s(const char * restrict format,
     va_list arg);
int vsnprintf_s(char * restrict s, rsize_t n,
     const char * restrict format,
     va_list arg);
int vsprintf_s(char * restrict s, rsize_t n,
     const char * restrict format,
     va_list arg);
int vsscanf_s(const char * restrict s,
     const char * restrict format,
     va_list arg);
char *gets_s(char *s, rsize_t n);
```

B.21 General utilities `<stdlib.h>`

```
size_t        ldiv_t        EXIT_FAILURE  MB_CUR_MAX
wchar_t       lldiv_t       EXIT_SUCCESS
div_t         NULL          RAND_MAX
```

```
double atof(const char *nptr);
int atoi(const char *nptr);
long int atol(const char *nptr);
long long int atoll(const char *nptr);
double strtod(const char * restrict nptr,
     char ** restrict endptr);
float strtof(const char * restrict nptr,
     char ** restrict endptr);
long double strtold(const char * restrict nptr,
     char ** restrict endptr);
long int strtol(const char * restrict nptr,
     char ** restrict endptr, int base);
long long int strtoll(const char * restrict nptr,
     char ** restrict endptr, int base);
unsigned long int strtoul(
     const char * restrict nptr,
     char ** restrict endptr, int base);
unsigned long long int strtoull(
     const char * restrict nptr,
     char ** restrict endptr, int base);
int rand(void);
void srand(unsigned int seed);
void *aligned_alloc(size_t alignment, size_t size);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
_Noreturn void abort(void);
int atexit(void (*func)(void));
int at_quick_exit(void (*func)(void));
_Noreturn void exit(int status);
_Noreturn void _Exit(int status);
char *getenv(const char *name);
_Noreturn void quick_exit(int status);
int system(const char *string);
```

```
void *bsearch(const void *key, const void *base,
     size_t nmemb, size_t size,
     int (*compar)(const void *, const void *));
void qsort(void *base, size_t nmemb, size_t size,
     int (*compar)(const void *, const void *));
int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer,
     long long int denom);
int mblen(const char *s, size_t n);
int mbtowc(wchar_t * restrict pwc,
     const char * restrict s, size_t n);
int wctomb(char *s, wchar_t wchar);
size_t mbstowcs(wchar_t * restrict pwcs,
     const char * restrict s, size_t n);
size_t wcstombs(char * restrict s,
     const wchar_t * restrict pwcs, size_t n);
```

```
__STDC_WANT_LIB_EXT1__
errno_t
rsize_t
constraint_handler_t
```

```
constraint_handler_t set_constraint_handler_s(
     constraint_handler_t handler);
void abort_handler_s(
     const char * restrict msg,
     void * restrict ptr,
     errno_t error);
void ignore_handler_s(
     const char * restrict msg,
     void * restrict ptr,
     errno_t error);
errno_t getenv_s(size_t * restrict len,
          char * restrict value, rsize_t maxsize,
          const char * restrict name);
```

```
void *bsearch_s(const void *key, const void *base,
     rsize_t nmemb, rsize_t size,
     int (*compar)(const void *k, const void *y,
                    void *context),
     void *context);
errno_t qsort_s(void *base, rsize_t nmemb, rsize_t size,
     int (*compar)(const void *x, const void *y,
                    void *context),
     void *context);
errno_t wctomb_s(int * restrict status,
     char * restrict s,
     rsize_t smax,
     wchar_t wc);
errno_t mbstowcs_s(size_t * restrict retval,
     wchar_t * restrict dst, rsize_t dstmax,
     const char * restrict src, rsize_t len);
errno_t wcstombs_s(size_t * restrict retval,
     char * restrict dst, rsize_t dstmax,
     const wchar_t * restrict src, rsize_t len);
```

B.22 `_Noreturn` `<stdnoreturn.h>`

```
noreturn
```

B.23 String handling `<string.h>`

```
size_t
NULL

void *memcpy(void * restrict s1,
     const void * restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strcpy(char * restrict s1,
     const char * restrict s2);
char *strncpy(char * restrict s1,
     const char * restrict s2, size_t n);
char *strcat(char * restrict s1,
     const char * restrict s2);
char *strncat(char * restrict s1,
     const char * restrict s2, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

```
size_t strxfrm(char * restrict s1,
     const char * restrict s2, size_t n);
void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char * restrict s1,
     const char * restrict s2);
void *memset(void *s, int c, size_t n);
char *strerror(int errnum);
size_t strlen(const char *s);

__STDC_WANT_LIB_EXT1__
errno_t
rsize_t

errno_t memcpy_s(void * restrict s1, rsize_t s1max,
     const void * restrict s2, rsize_t n);
errno_t memmove_s(void *s1, rsize_t s1max,
     const void *s2, rsize_t n);
errno_t strcpy_s(char * restrict s1,
     rsize_t s1max,
     const char * restrict s2);
errno_t strncpy_s(char * restrict s1,
     rsize_t s1max,
     const char * restrict s2,
     rsize_t n);
errno_t strcat_s(char * restrict s1,
     rsize_t s1max,
     const char * restrict s2);
errno_t strncat_s(char * restrict s1,
     rsize_t s1max,
     const char * restrict s2,
     rsize_t n);
char *strtok_s(char * restrict s1,
     rsize_t * restrict s1max,
     const char * restrict s2,
     char ** restrict ptr);
errno_t memset_s(void *s, rsize_t smax, int c, rsize_t n)
```

```
errno_t strerror_s(char *s, rsize_t maxsize,
     errno_t errnum);
size_t strerrorlen_s(errno_t errnum);
size_t strnlen_s(const char *s, size_t maxsize);
```

B.24 Type-generic math `<tgmath.h>`

```
acos        sqrt        fmod        nextafter
asin        fabs        frexp       nexttoward
atan        atan2       hypot       remainder
acosh       cbrt        ilogb       remquo
asinh       ceil        ldexp       rint
atanh       copysign    lgamma      round
cos         erf         llrint      scalbn
sin         erfc        llround     scalbln
tan         exp2        log10       tgamma
cosh        expm1       log1p       trunc
sinh        fdim        log2        carg
tanh        floor       logb        cimag
exp         fma         lrint       conj
log         fmax        lround      cproj
pow         fmin        nearbyint   creal
```

B.25 Threads `<threads.h>`

```
thread_local                 once_flag
ONCE_FLAG_INIT               mtx_plain
TSS_DTOR_ITERATIONS          mtx_recursive
cnd_t                        mtx_timed
thrd_t                       thrd_timedout
tss_t                        thrd_success
mtx_t                        thrd_busy
tss_dtor_t                   thrd_error
thrd_start_t                 thrd_nomem
```

```
void call_once(once_flag *flag, void (*func)(void));
int cnd_broadcast(cnd_t *cond);
void cnd_destroy(cnd_t *cond);
int cnd_init(cnd_t *cond);
int cnd_signal(cnd_t *cond);
int cnd_timedwait(cnd_t *restrict cond,
     mtx_t *restrict mtx,
     const struct timespec *restrict ts);
int cnd_wait(cnd_t *cond, mtx_t *mtx);
```

```
void mtx_destroy(mtx_t *mtx);
int mtx_init(mtx_t *mtx, int type);
int mtx_lock(mtx_t *mtx);
int mtx_timedlock(mtx_t *restrict mtx,
     const struct timespec *restrict ts);
int mtx_trylock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
int thrd_create(thrd_t *thr, thrd_start_t func,
     void *arg);
thrd_t thrd_current(void);
int thrd_detach(thrd_t thr);
int thrd_equal(thrd_t thr0, thrd_t thr1);
_Noreturn void thrd_exit(int res);
int thrd_join(thrd_t thr, int *res);
int thrd_sleep(const struct timespec *duration,
     struct timespec *remaining);
void thrd_yield(void);
int tss_create(tss_t *key, tss_dtor_t dtor);
void tss_delete(tss_t key);
void *tss_get(tss_t key);
int tss_set(tss_t key, void *val);
```

B.26 Date and time <time.h>

```
NULL               size_t             struct timespec
CLOCKS_PER_SEC     clock_t            struct tm
TIME_UTC           time_t
```

```
clock_t clock(void);
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm *timeptr);
time_t time(time_t *timer);
int timespec_get(timespec *ts, int base);
char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timer);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
size_t strftime(char * restrict s,
     size_t maxsize,
     const char * restrict format,
     const struct tm * restrict timeptr);
```

```
__STDC_WANT_LIB_EXT1__
```

```
errno_t
rsize_t
```

```
errno_t asctime_s(char *s, rsize_t maxsize,
     const struct tm *timeptr);
errno_t ctime_s(char *s, rsize_t maxsize,
     const time_t *timer);
struct tm *gmtime_s(const time_t * restrict timer,
     struct tm * restrict result);
struct tm *localtime_s(const time_t * restrict timer,
     struct tm * restrict result);
```

B.27 Unicode utilities `<uchar.h>`

```
mbstate_t     size_t        char16_t      char32_t
```

```
size_t mbrtoc16(char16_t * restrict pc16,
     const char * restrict s, size_t n,
     mbstate_t * restrict ps);
size_t c16rtomb(char * restrict s, char16_t c16,
     mbstate_t * restrict ps);
size_t mbrtoc32(char32_t * restrict pc32,
     const char * restrict s, size_t n,
     mbstate_t * restrict ps);
size_t c32rtomb(char * restrict s, char32_t c32,
     mbstate_t * restrict ps);
```

B.28 Extended multibyte/wide character utilities `<wchar.h>`

```
wchar_t            wint_t             WCHAR_MAX
size_t             struct tm          WCHAR_MIN
mbstate_t          NULL               WEOF
```

```
int fwprintf(FILE * restrict stream,
     const wchar_t * restrict format, ...);
int fwscanf(FILE * restrict stream,
     const wchar_t * restrict format, ...);
int swprintf(wchar_t * restrict s, size_t n,
     const wchar_t * restrict format, ...);
int swscanf(const wchar_t * restrict s,
     const wchar_t * restrict format, ...);
int vfwprintf(FILE * restrict stream,
     const wchar_t * restrict format, va_list arg);
```

```
int vfwscanf(FILE * restrict stream,
     const wchar_t * restrict format, va_list arg);
int vswprintf(wchar_t * restrict s, size_t n,
     const wchar_t * restrict format, va_list arg);
int vswscanf(const wchar_t * restrict s,
     const wchar_t * restrict format, va_list arg);
int vwprintf(const wchar_t * restrict format,
     va_list arg);
int vwscanf(const wchar_t * restrict format,
     va_list arg);
int wprintf(const wchar_t * restrict format, ...);
int wscanf(const wchar_t * restrict format, ...);
wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t * restrict s, int n,
     FILE * restrict stream);
wint_t fputwc(wchar_t c, FILE *stream);
int fputws(const wchar_t * restrict s,
     FILE * restrict stream);
int fwide(FILE *stream, int mode);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c);
wint_t ungetwc(wint_t c, FILE *stream);
double wcstod(const wchar_t * restrict nptr,
     wchar_t ** restrict endptr);
float wcstof(const wchar_t * restrict nptr,
     wchar_t ** restrict endptr);
long double wcstold(const wchar_t * restrict nptr,
     wchar_t ** restrict endptr);
long int wcstol(const wchar_t * restrict nptr,
     wchar_t ** restrict endptr, int base);
long long int wcstoll(const wchar_t * restrict nptr,
     wchar_t ** restrict endptr, int base);
unsigned long int wcstoul(const wchar_t * restrict nptr,
     wchar_t ** restrict endptr, int base);
unsigned long long int wcstoull(
     const wchar_t * restrict nptr,
     wchar_t ** restrict endptr, int base);
```

```
wchar_t *wcscpy(wchar_t * restrict s1,
     const wchar_t * restrict s2);
wchar_t *wcsncpy(wchar_t * restrict s1,
     const wchar_t * restrict s2, size_t n);
wchar_t *wmemcpy(wchar_t * restrict s1,
     const wchar_t * restrict s2, size_t n);
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2,
     size_t n);
wchar_t *wcscat(wchar_t * restrict s1,
     const wchar_t * restrict s2);
wchar_t *wcsncat(wchar_t * restrict s1,
     const wchar_t * restrict s2, size_t n);
int wcscmp(const wchar_t *s1, const wchar_t *s2);
int wcscoll(const wchar_t *s1, const wchar_t *s2);
int wcsncmp(const wchar_t *s1, const wchar_t *s2,
     size_t n);
size_t wcsxfrm(wchar_t * restrict s1,
     const wchar_t * restrict s2, size_t n);
int wmemcmp(const wchar_t *s1, const wchar_t *s2,
     size_t n);
wchar_t *wcschr(const wchar_t *s, wchar_t c);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcstok(wchar_t * restrict s1,
     const wchar_t * restrict s2,
     wchar_t ** restrict ptr);
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
size_t wcslen(const wchar_t *s);
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
size_t wcsftime(wchar_t * restrict s, size_t maxsize,
     const wchar_t * restrict format,
     const struct tm * restrict timeptr);
wint_t btowc(int c);
int wctob(wint_t c);
int mbsinit(const mbstate_t *ps);
size_t mbrlen(const char * restrict s, size_t n,
     mbstate_t * restrict ps);
```

```
size_t mbrtowc(wchar_t * restrict pwc,
     const char * restrict s, size_t n,
     mbstate_t * restrict ps);
size_t wcrtomb(char * restrict s, wchar_t wc,
     mbstate_t * restrict ps);
size_t mbsrtowcs(wchar_t * restrict dst,
     const char ** restrict src, size_t len,
     mbstate_t * restrict ps);
size_t wcsrtombs(char * restrict dst,
     const wchar_t ** restrict src, size_t len,
     mbstate_t * restrict ps);

__STDC_WANT_LIB_EXT1__
errno_t
rsize_t

int fwprintf_s(FILE * restrict stream,
     const wchar_t * restrict format, ...);
int fwscanf_s(FILE * restrict stream,
     const wchar_t * restrict format, ...);
int snwprintf_s(wchar_t * restrict s,
     rsize_t n,
     const wchar_t * restrict format, ...);
int swprintf_s(wchar_t * restrict s, rsize_t n,
     const wchar_t * restrict format, ...);
int swscanf_s(const wchar_t * restrict s,
     const wchar_t * restrict format, ...);
int vfwprintf_s(FILE * restrict stream,
     const wchar_t * restrict format,
     va_list arg);
int vfwscanf_s(FILE * restrict stream,
     const wchar_t * restrict format, va_list arg);
int vsnwprintf_s(wchar_t * restrict s,
     rsize_t n,
     const wchar_t * restrict format,
     va_list arg);
int vswprintf_s(wchar_t * restrict s,
     rsize_t n,
     const wchar_t * restrict format,
     va_list arg);
```

```
int vswscanf_s(const wchar_t * restrict s,
     const wchar_t * restrict format,
     va_list arg);
int vwprintf_s(const wchar_t * restrict format,
     va_list arg);
int vwscanf_s(const wchar_t * restrict format,
     va_list arg);
int wprintf_s(const wchar_t * restrict format, ...);
int wscanf_s(const wchar_t * restrict format, ...);
errno_t wcscpy_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2);
errno_t wcsncpy_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2,
     rsize_t n);
errno_t wmemcpy_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2,
     rsize_t n);
errno_t wmemmove_s(wchar_t *s1, rsize_t s1max,
     const wchar_t *s2, rsize_t n);
errno_t wcscat_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2);
errno_t wcsncat_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2,
     rsize_t n);
wchar_t *wcstok_s(wchar_t * restrict s1,
     rsize_t * restrict s1max,
     const wchar_t * restrict s2,
     wchar_t ** restrict ptr);
size_t wcsnlen_s(const wchar_t *s, size_t maxsize);
errno_t wcrtomb_s(size_t * restrict retval,
     char * restrict s, rsize_t smax,
     wchar_t wc, mbstate_t * restrict ps);
```

```
errno_t mbsrtowcs_s(size_t * restrict retval,
     wchar_t * restrict dst, rsize_t dstmax,
     const char ** restrict src, rsize_t len,
     mbstate_t * restrict ps);
errno_t wcsrtombs_s(size_t * restrict retval,
     char * restrict dst, rsize_t dstmax,
     const wchar_t ** restrict src, rsize_t len,
     mbstate_t * restrict ps);
```

B.29 Wide character classification and mapping utilities `<wctype.h>`

```
wint_t          wctrans_t      wctype_t       WEOF
```

```
int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswblank(wint_t wc);
int iswcntrl(wint_t wc);
int iswdigit(wint_t wc);
int iswgraph(wint_t wc);
int iswlower(wint_t wc);
int iswprint(wint_t wc);
int iswpunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswxdigit(wint_t wc);
int iswctype(wint_t wc, wctype_t desc);
wctype_t wctype(const char *property);
wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
wint_t towctrans(wint_t wc, wctrans_t desc);
wctrans_t wctrans(const char *property);
```

Annex C
(informative)

Sequence points

1 The following are the sequence points described in 5.1.2.3:

— Between the evaluations of the function designator and actual arguments in a function call and the actual call. (6.5.2.2).

— Between the evaluations of the first and second operands of the following operators: logical AND **`&&`** (6.5.13); logical OR `||` (6.5.14); comma **`,`** (6.5.17).

— Between the evaluations of the first operand of the conditional **`? :`** operator and whichever of the second and third operands is evaluated (6.5.15).

— The end of a full declarator: declarators (6.7.6);

— Between the evaluation of a full expression and the next full expression to be evaluated. The following are full expressions: an initializer that is not part of a compound literal (6.7.9); the expression in an expression statement (6.8.3); the controlling expression of a selection statement (**`if`** or **`switch`**) (6.8.4); the controlling expression of a **`while`** or **`do`** statement (6.8.5); each of the (optional) expressions of a **`for`** statement (6.8.5.3); the (optional) expression in a **`return`** statement (6.8.6.4).

— Immediately before a library function returns (7.1.4).

— After the actions associated with each formatted input/output function conversion specifier (7.21.6, 7.29.2).

— Immediately before and immediately after each call to a comparison function, and also between any call to a comparison function and any movement of the objects passed as arguments to that call (7.22.5).

Annex D

(normative)

Universal character names for identifiers

1 This clause lists the hexadecimal code values that are valid in universal character names in identifiers.

D.1 Ranges of characters allowed

1 00A8, 00AA, 00AD, 00AF, 00B2−00B5, 00B7−00BA, 00BC−00BE, 00C0−00D6, 00D8−00F6, 00F8−00FF

2 0100−167F, 1681−180D, 180F−1FFF

3 200B−200D, 202A−202E, 203F−2040, 2054, 2060−206F

4 2070−218F, 2460−24FF, 2776−2793, 2C00−2DFF, 2E80−2FFF

5 3004−3007, 3021−302F, 3031−303F

6 3040−D7FF

7 F900−FD3D, FD40−FDCF, FDF0−FE44, FE47−FFFD

8 10000−1FFFD, 20000−2FFFD, 30000−3FFFD, 40000−4FFFD, 50000−5FFFD, 60000−6FFFD, 70000−7FFFD, 80000−8FFFD, 90000−9FFFD, A0000−AFFFD, B0000−BFFFD, C0000−CFFFD, D0000−DFFFD, E0000−EFFFD

D.2 Ranges of characters disallowed initially

1 0300−036F, 1DC0−1DFF, 20D0−20FF, FE20−FE2F

Annex E

(informative)

Implementation limits

1 The contents of the header **<limits.h>** are given below, in alphabetical order. The minimum magnitudes shown shall be replaced by implementation-defined magnitudes with the same sign. The values shall all be constant expressions suitable for use in **#if** preprocessing directives. The components are described further in 5.2.4.2.1.

```
#define CHAR_BIT                         8
#define CHAR_MAX    UCHAR_MAX or SCHAR_MAX
#define CHAR_MIN            0 or SCHAR_MIN
#define INT_MAX                     +32767
#define INT_MIN                     -32767
#define LONG_MAX               +2147483647
#define LONG_MIN               -2147483647
#define LLONG_MAX     +9223372036854775807
#define LLONG_MIN     -9223372036854775807
#define MB_LEN_MAX                       1
#define SCHAR_MAX                     +127
#define SCHAR_MIN                     -127
#define SHRT_MAX                    +32767
#define SHRT_MIN                    -32767
#define UCHAR_MAX                      255
#define USHRT_MAX                    65535
#define UINT_MAX                     65535
#define ULONG_MAX               4294967295
#define ULLONG_MAX    18446744073709551615
```

2 The contents of the header **<float.h>** are given below. All integer values, except **FLT_ROUNDS**, shall be constant expressions suitable for use in **#if** preprocessing directives; all floating values shall be constant expressions. The components are described further in 5.2.4.2.2.

3 The values given in the following list shall be replaced by implementation-defined expressions:

```
#define FLT_EVAL_METHOD
#define FLT_ROUNDS
```

4 The values given in the following list shall be replaced by implementation-defined constant expressions that are greater or equal in magnitude (absolute value) to those shown, with the same sign:

```
#define DLB_DECIMAL_DIG                     10
#define DBL_DIG                             10
#define DBL_MANT_DIG
#define DBL_MAX_10_EXP                     +37
#define DBL_MAX_EXP
#define DBL_MIN_10_EXP                     -37
#define DBL_MIN_EXP
#define DECIMAL_DIG                         10
#define FLT_DECIMAL_DIG                      6
#define FLT_DIG                              6
#define FLT_MANT_DIG
#define FLT_MAX_10_EXP                     +37
#define FLT_MAX_EXP
#define FLT_MIN_10_EXP                     -37
#define FLT_MIN_EXP
#define FLT_RADIX                            2
#define LDLB_DECIMAL_DIG                    10
#define LDBL_DIG                            10
#define LDBL_MANT_DIG
#define LDBL_MAX_10_EXP                    +37
#define LDBL_MAX_EXP
#define LDBL_MIN_10_EXP                    -37
#define LDBL_MIN_EXP
```

5 The values given in the following list shall be replaced by implementation-defined constant expressions with values that are greater than or equal to those shown:

```
#define DBL_MAX                          1E+37
#define FLT_MAX                          1E+37
#define LDBL_MAX                         1E+37
```

6 The values given in the following list shall be replaced by implementation-defined constant expressions with (positive) values that are less than or equal to those shown:

```
#define DBL_EPSILON                       1E-9
#define DBL_MIN                          1E-37
#define FLT_EPSILON                       1E-5
#define FLT_MIN                          1E-37
#define LDBL_EPSILON                      1E-9
#define LDBL_MIN                         1E-37
```

Annex F

(normative)

IEC 60559 floating-point arithmetic

F.1 Introduction

1 This annex specifies C language support for the IEC 60559 floating-point standard. The *IEC 60559 floating-point standard* is specifically *Binary floating-point arithmetic for microprocessor systems, second edition* (IEC 60559:1989), previously designated IEC 559:1989 and as *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE 754−1985). *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE 854−1987) generalizes the binary standard to remove dependencies on radix and word length. *IEC 60559* generally refers to the floating-point standard, as in IEC 60559 operation, IEC 60559 format, etc. An implementation that defines **`__STDC_IEC_559__`** shall conform to the specifications in this annex.[356)] Where a binding between the C language and IEC 60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless stated otherwise. Since negative and positive infinity are representable in IEC 60559 formats, all real numbers lie within the range of representable values.

F.2 Types

1 The C floating types match the IEC 60559 formats as follows:

— The **`float`** type matches the IEC 60559 single format.

— The **`double`** type matches the IEC 60559 double format.

— The **`long double`** type matches an IEC 60559 extended format,[357)] else a non-IEC 60559 extended format, else the IEC 60559 **`double`** format.

Any non-IEC 60559 extended format used for the **`long double`** type shall have more precision than IEC 60559 double and at least the range of IEC 60559 double.[358)]

---

356) Implementations that do not define **`__STDC_IEC_559__`** are not required to conform to these specifications.

357) ‘‘Extended’’ is IEC 60559’s double-extended data format. Extended refers to both the common 80-bit and quadruple 128-bit IEC 60559 formats.

358) A non-IEC 60559 **`long double`** type is required to provide infinity and NaNs, as its values include all **`double`** values.

Recommended practice

2 The **long double** type should match an IEC 60559 extended format.

F.2.1 Infinities, signed zeros, and NaNs

1 This specification does not define the behavior of signaling NaNs.[359] It generally uses the term *NaN* to denote quiet NaNs. The **NAN** and **INFINITY** macros and the **nan** functions in **<math.h>** provide designations for IEC 60559 NaNs and infinities.

F.3 Operators and functions

1 C operators and functions provide IEC 60559 required and recommended facilities as listed below.

— The **+**, **-**, *****, and **/** operators provide the IEC 60559 add, subtract, multiply, and divide operations.

— The **sqrt** functions in **<math.h>** provide the IEC 60559 square root operation.

— The **remainder** functions in **<math.h>** provide the IEC 60559 remainder operation. The **remquo** functions in **<math.h>** provide the same operation but with additional information.

— The **rint** functions in **<math.h>** provide the IEC 60559 operation that rounds a floating-point number to an integer value (in the same precision). The **nearbyint** functions in **<math.h>** provide the nearbyinteger function recommended in the Appendix to ANSI/IEEE 854.

— The conversions for floating types provide the IEC 60559 conversions between floating-point precisions.

— The conversions from integer to floating types provide the IEC 60559 conversions from integer to floating point.

— The conversions from floating to integer types provide IEC 60559-like conversions but always round toward zero.

— The **lrint** and **llrint** functions in **<math.h>** provide the IEC 60559 conversions, which honor the directed rounding mode, from floating point to the **long int** and **long long int** integer formats. The **lrint** and **llrint** functions can be used to implement IEC 60559 conversions from floating to other integer formats.

— The translation time conversion of floating constants and the **strtod**, **strtof**, **strtold**, **fprintf**, **fscanf**, and related library functions in **<stdlib.h>**,

---

359) Since NaNs created by IEC 60559 operations are always quiet, quiet NaNs (along with infinities) are sufficient for closure of the arithmetic.

<stdio.h>, and **<wchar.h>** provide IEC 60559 binary-decimal conversions. The **strtold** function in **<stdlib.h>** provides the conv function recommended in the Appendix to ANSI/IEEE 854.

— The relational and equality operators provide IEC 60559 comparisons. IEC 60559 identifies a need for additional comparison predicates to facilitate writing code that accounts for NaNs. The comparison macros (**isgreater**, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**, and **isunordered**) in **<math.h>** supplement the language operators to address this need. The **islessgreater** and **isunordered** macros provide respectively a quiet version of the <> predicate and the unordered predicate recommended in the Appendix to IEC 60559.

— The **feclearexcept**, **feraiseexcept**, and **fetestexcept** functions in **<fenv.h>** provide the facility to test and alter the IEC 60559 floating-point exception status flags. The **fegetexceptflag** and **fesetexceptflag** functions in **<fenv.h>** provide the facility to save and restore all five status flags at one time. These functions are used in conjunction with the type **fexcept_t** and the floating-point exception macros (**FE_INEXACT**, **FE_DIVBYZERO**, **FE_UNDERFLOW**, **FE_OVERFLOW**, **FE_INVALID**) also in **<fenv.h>**.

— The **fegetround** and **fesetround** functions in **<fenv.h>** provide the facility to select among the IEC 60559 directed rounding modes represented by the rounding direction macros in **<fenv.h>** (**FE_TONEAREST**, **FE_UPWARD**, **FE_DOWNWARD**, **FE_TOWARDZERO**) and the values **0**, **1**, **2**, and **3** of **FLT_ROUNDS** are the IEC 60559 directed rounding modes.

— The **fegetenv**, **feholdexcept**, **fesetenv**, and **feupdateenv** functions in **<fenv.h>** provide a facility to manage the floating-point environment, comprising the IEC 60559 status flags and control modes.

— The **copysign** functions in **<math.h>** provide the copysign function recommended in the Appendix to IEC 60559.

— The **fabs** functions in **<math.h>** provide the abs function recommended in the Appendix to IEC 60559.

— The unary minus (**-**) operator provides the unary minus (−) operation recommended in the Appendix to IEC 60559.

— The **scalbn** and **scalbln** functions in **<math.h>** provide the scalb function recommended in the Appendix to IEC 60559.

— The **logb** functions in **<math.h>** provide the logb function recommended in the Appendix to IEC 60559, but following the newer specifications in ANSI/IEEE 854.

— The **nextafter** and **nexttoward** functions in **<math.h>** provide the nextafter function recommended in the Appendix to IEC 60559 (but with a minor change to better handle signed zeros).

— The **isfinite** macro in **<math.h>** provides the finite function recommended in the Appendix to IEC 60559.

— The **isnan** macro in **<math.h>** provides the isnan function recommended in the Appendix to IEC 60559.

— The **signbit** macro and the **fpclassify** macro in **<math.h>**, used in conjunction with the number classification macros (**FP_NAN**, **FP_INFINITE**, **FP_NORMAL**, **FP_SUBNORMAL**, **FP_ZERO**), provide the facility of the class function recommended in the Appendix to IEC 60559 (except that the classification macros defined in 7.12.3 do not distinguish signaling from quiet NaNs).

F.4 Floating to integer conversion

1 If the integer type is **_Bool**, 6.3.1.2 applies and no floating-point exceptions are raised (even for NaN). Otherwise, if the floating value is infinite or NaN or if the integral part of the floating value exceeds the range of the integer type, then the ‘‘invalid’’ floating-point exception is raised and the resulting value is unspecified. Otherwise, the resulting value is determined by 6.3.1.4. Conversion of an integral floating value that does not exceed the range of the integer type raises no floating-point exceptions; whether conversion of a non-integral floating value raises the ‘‘inexact’’ floating-point exception is unspecified.[360)]

F.5 Binary-decimal conversion

1 Conversion from the widest supported IEC 60559 format to decimal with **DECIMAL_DIG** digits and back is the identity function.[361)]

2 Conversions involving IEC 60559 formats follow all pertinent recommended practice. In particular, conversion between any supported IEC 60559 format and decimal with **DECIMAL_DIG** or fewer significant digits is correctly rounded (honoring the current rounding mode), which assures that conversion from the widest supported IEC 60559 format to decimal with **DECIMAL_DIG** digits and back is the identity function.

---

360) ANSI/IEEE 854, but not IEC 60559 (ANSI/IEEE 754), directly specifies that floating-to-integer conversions raise the ‘‘inexact’’ floating-point exception for non-integer in-range values. In those cases where it matters, library functions can be used to effect such conversions with or without raising the ‘‘inexact’’ floating-point exception. See **rint**, **lrint**, **llrint**, and **nearbyint** in **<math.h>**.

361) If the minimum-width IEC 60559 extended format (64 bits of precision) is supported, **DECIMAL_DIG** shall be at least 21. If IEC 60559 double (53 bits of precision) is the widest IEC 60559 format supported, then **DECIMAL_DIG** shall be at least 17. (By contrast, **LDBL_DIG** and **DBL_DIG** are 18 and 15, respectively, for these formats.)

3 Functions such as **`strtod`** that convert character sequences to floating types honor the rounding direction. Hence, if the rounding direction might be upward or downward, the implementation cannot convert a minus-signed sequence by negating the converted unsigned sequence.

F.6 The `return` statement

If the return expression is evaluated in a floating-point format different from the return type, the expression is converted as if by assignment[362)] to the return type of the function and the resulting value is returned to the caller.

F.7 Contracted expressions

1 A contracted expression is correctly rounded (once) and treats infinities, NaNs, signed zeros, subnormals, and the rounding directions in a manner consistent with the basic arithmetic operations covered by IEC 60559.

Recommended practice

2 A contracted expression should raise floating-point exceptions in a manner generally consistent with the basic arithmetic operations.

F.8 Floating-point environment

1 The floating-point environment defined in **`<fenv.h>`** includes the IEC 60559 floating-point exception status flags and directed-rounding control modes. It includes also IEC 60559 dynamic rounding precision and trap enablement modes, if the implementation supports them.[363)]

F.8.1 Environment management

1 IEC 60559 requires that floating-point operations implicitly raise floating-point exception status flags, and that rounding control modes can be set explicitly to affect result values of floating-point operations. When the state for the **`FENV_ACCESS`** pragma (defined in **`<fenv.h>`**) is ‘‘on’’, these changes to the floating-point state are treated as side effects which respect sequence points.[364)]

---

362) Assignment removes any extra range and precision.

363) This specification does not require dynamic rounding precision nor trap enablement modes.

364) If the state for the **`FENV_ACCESS`** pragma is ‘‘off’’, the implementation is free to assume the floating-point control modes will be the default ones and the floating-point status flags will not be tested, which allows certain optimizations (see F.9).

F.8.2 Translation

1 During translation the IEC 60559 default modes are in effect:

— The rounding direction mode is rounding to nearest.

— The rounding precision mode (if supported) is set so that results are not shortened.

— Trapping or stopping (if supported) is disabled on all floating-point exceptions.

Recommended practice

2 The implementation should produce a diagnostic message for each translation-time floating-point exception, other than ‘‘inexact’’;[365)] the implementation should then proceed with the translation of the program.

F.8.3 Execution

1 At program startup the floating-point environment is initialized as prescribed by IEC 60559:

— All floating-point exception status flags are cleared.

— The rounding direction mode is rounding to nearest.

— The dynamic rounding precision mode (if supported) is set so that results are not shortened.

— Trapping or stopping (if supported) is disabled on all floating-point exceptions.

F.8.4 Constant expressions

1 An arithmetic constant expression of floating type, other than one in an initializer for an object that has static or thread storage duration, is evaluated (as if) during execution; thus, it is affected by any operative floating-point control modes and raises floating-point exceptions as required by IEC 60559 (provided the state for the **`FENV_ACCESS`** pragma is ‘‘on’’).[366)]

2 EXAMPLE

365) As floating constants are converted to appropriate internal representations at translation time, their conversion is subject to default rounding modes and raises no execution-time floating-point exceptions (even where the state of the **`FENV_ACCESS`** pragma is ‘‘on’’). Library functions, for example **`strtod`**, provide execution-time conversion of numeric strings.

366) Where the state for the **`FENV_ACCESS`** pragma is ‘‘on’’, results of inexact expressions like **`1.0/3.0`** are affected by rounding modes set at execution time, and expressions such as **`0.0/0.0`** and **`1.0/0.0`** generate execution-time floating-point exceptions. The programmer can achieve the efficiency of translation-time evaluation through static initialization, such as

```
const static double one_third = 1.0/3.0;
```

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(void)
{
      float w[] = { 0.0/0.0 };   // raises an exception
      static float x = 0.0/0.0;  // does not raise an exception
      float y = 0.0/0.0;         // raises an exception
      double z = 0.0/0.0;        // raises an exception
      /* ... */
}
```

3 For the static initialization, the division is done at translation time, raising no (execution-time) floating-point exceptions. On the other hand, for the three automatic initializations the invalid division occurs at execution time.

F.8.5 Initialization

1 All computation for automatic initialization is done (as if) at execution time; thus, it is affected by any operative modes and raises floating-point exceptions as required by IEC 60559 (provided the state for the **FENV_ACCESS** pragma is ‘‘on’’). All computation for initialization of objects that have static or thread storage duration is done (as if) at translation time.

2 EXAMPLE

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(void)
{
      float u[] = { 1.1e75 };    // raises exceptions
      static float v = 1.1e75;   // does not raise exceptions
      float w = 1.1e75;          // raises exceptions
      double x = 1.1e75;         // may raise exceptions
      float y = 1.1e75f;         // may raise exceptions
      long double z = 1.1e75;    // does not raise exceptions
      /* ... */
}
```

3 The static initialization of **v** raises no (execution-time) floating-point exceptions because its computation is done at translation time. The automatic initialization of **u** and **w** require an execution-time conversion to **float** of the wider value **1.1e75**, which raises floating-point exceptions. The automatic initializations of **x** and **y** entail execution-time conversion; however, in some expression evaluation methods, the conversions is not to a narrower format, in which case no floating-point exception is raised.[367] The automatic initialization of **z** entails execution-time conversion, but not to a narrower format, so no floating-point exception is raised. Note that the conversions of the floating constants **1.1e75** and **1.1e75f** to

367) Use of **float_t** and **double_t** variables increases the likelihood of translation-time computation. For example, the automatic initialization

```
double_t x = 1.1e75;
```

could be done at translation time, regardless of the expression evaluation method.

their internal representations occur at translation time in all cases.

F.8.6 Changing the environment

1 Operations defined in 6.5 and functions and macros defined for the standard libraries change floating-point status flags and control modes just as indicated by their specifications (including conformance to IEC 60559). They do not change flags or modes (so as to be detectable by the user) in any other cases.

2 If the argument to the **`feraiseexcept`** function in **`<fenv.h>`** represents IEC 60559 valid coincident floating-point exceptions for atomic operations (namely ‘‘overflow’’ and ‘‘inexact’’, or ‘‘underflow’’ and ‘‘inexact’’), then ‘‘overflow’’ or ‘‘underflow’’ is raised before ‘‘inexact’’.

F.9 Optimization

1 This section identifies code transformations that might subvert IEC 60559-specified behavior, and others that do not.

F.9.1 Global transformations

1 Floating-point arithmetic operations and external function calls may entail side effects which optimization shall honor, at least where the state of the **`FENV_ACCESS`** pragma is ‘‘on’’. The flags and modes in the floating-point environment may be regarded as global variables; floating-point operations (**`+`**, **`*`**, etc.) implicitly read the modes and write the flags.

2 Concern about side effects may inhibit code motion and removal of seemingly useless code. For example, in

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(double x)
{
     /* ... */
     for (i = 0; i < n; i++) x + 1;
     /* ... */
}
```

`x + 1` might raise floating-point exceptions, so cannot be removed. And since the loop body might not execute (maybe $0 \geq$ **`n`**), **`x + 1`** cannot be moved out of the loop. (Of course these optimizations are valid if the implementation can rule out the nettlesome cases.)

3 This specification does not require support for trap handlers that maintain information about the order or count of floating-point exceptions. Therefore, between function calls, floating-point exceptions need not be precise: the actual order and number of occurrences of floating-point exceptions (> 1) may vary from what the source code expresses. Thus,

the preceding loop could be treated as

```
if (0 < n) x + 1;
```

F.9.2 Expression transformations

1 $x/2 \leftrightarrow x \times 0.5$ Although similar transformations involving inexact constants generally do not yield numerically equivalent expressions, if the constants are exact then such transformations can be made on IEC 60559 machines and others that round perfectly.

$1 \times x$ and $x/1 \rightarrow x$ The expressions $1 \times x$, $x/1$, and x are equivalent (on IEC 60559 machines, among others).[368)]

$x/x \rightarrow 1.0$ The expressions x/x and 1.0 are not equivalent if x can be zero, infinite, or NaN.

$x - y \leftrightarrow x + (-y)$ The expressions $x - y$, $x + (-y)$, and $(-y) + x$ are equivalent (on IEC 60559 machines, among others).

$x - y \leftrightarrow -(y - x)$ The expressions $x - y$ and $-(y - x)$ are not equivalent because $1 - 1$ is +0 but $-(1 - 1)$ is −0 (in the default rounding direction).[369)]

$x - x \rightarrow 0.0$ The expressions $x - x$ and 0.0 are not equivalent if x is a NaN or infinite.

$0 \times x \rightarrow 0.0$ The expressions $0 \times x$ and 0.0 are not equivalent if x is a NaN, infinite, or −0.

$x + 0 \rightarrow x$ The expressions $x + 0$ and x are not equivalent if x is −0, because $(-0) + (+0)$ yields +0 (in the default rounding direction), not −0.

$x - 0 \rightarrow x$ $(+0) - (+0)$ yields −0 when rounding is downward (toward $-\infty$), but +0 otherwise, and $(-0) - (+0)$ always yields −0; so, if the state of the **FENV_ACCESS** pragma is ‘‘off’’, promising default rounding, then the implementation can replace $x - 0$ by x, even if x might be zero.

$-x \leftrightarrow 0 - x$ The expressions $-x$ and $0 - x$ are not equivalent if x is +0, because $-(+0)$ yields −0, but $0 - (+0)$ yields +0 (unless rounding is downward).

368) Strict support for signaling NaNs — not required by this specification — would invalidate these and other transformations that remove arithmetic operators.

369) IEC 60559 prescribes a signed zero to preserve mathematical identities across certain discontinuities. Examples include:

$1/(1/\pm\infty)$ is $\pm\infty$

and

conj(csqrt(z)) is csqrt(conj(z)),

for complex z.

F.9.3 Relational operators

1 $x \neq x \to$ false The expression $x \neq x$ is true if x is a NaN.

$x = x \to$ true The expression $x = x$ is false if x is a NaN.

$x < y \to$ **isless(***x***,***y***)** (and similarly for ≤, >, ≥) Though numerically equal, these expressions are not equivalent because of side effects when x or y is a NaN and the state of the **FENV_ACCESS** pragma is ‘‘on’’. This transformation, which would be desirable if extra code were required to cause the ‘‘invalid’’ floating-point exception for unordered cases, could be performed provided the state of the **FENV_ACCESS** pragma is ‘‘off’’.

The sense of relational operators shall be maintained. This includes handling unordered cases as expressed by the source code.

2 EXAMPLE

```
// calls g and raises "invalid" if a and b are unordered
if (a < b)
      f();
else
      g();
```

is not equivalent to

```
// calls f and raises "invalid" if a and b are unordered
if (a >= b)
      g();
else
      f();
```

nor to

```
// calls f without raising "invalid" if a and b are unordered
if (isgreaterequal(a,b))
      g();
else
      f();
```

nor, unless the state of the **FENV_ACCESS** pragma is ‘‘off’’, to

```
// calls g without raising "invalid" if a and b are unordered
if (isless(a,b))
      f();
else
      g();
```

but is equivalent to

```
if (!(a < b))
      g();
else
      f();
```

F.9.4 Constant arithmetic

1 The implementation shall honor floating-point exceptions raised by execution-time constant arithmetic wherever the state of the **FENV_ACCESS** pragma is ‘‘on’’. (See F.8.4 and F.8.5.) An operation on constants that raises no floating-point exception can be folded during translation, except, if the state of the **FENV_ACCESS** pragma is ‘‘on’’, a further check is required to assure that changing the rounding direction to downward does not alter the sign of the result,[370] and implementations that support dynamic rounding precision modes shall assure further that the result of the operation raises no floating-point exception when converted to the semantic type of the operation.

F.10 Mathematics **<math.h>**

1 This subclause contains specifications of **<math.h>** facilities that are particularly suited for IEC 60559 implementations.

2 The Standard C macro **HUGE_VAL** and its **float** and **long double** analogs, **HUGE_VALF** and **HUGE_VALL**, expand to expressions whose values are positive infinities.

3 Special cases for functions in **<math.h>** are covered directly or indirectly by IEC 60559. The functions that IEC 60559 specifies directly are identified in F.3. The other functions in **<math.h>** treat infinities, NaNs, signed zeros, subnormals, and (provided the state of the **FENV_ACCESS** pragma is ‘‘on’’) the floating-point status flags in a manner consistent with the basic arithmetic operations covered by IEC 60559.

4 The expression **math_errhandling & MATH_ERREXCEPT** shall evaluate to a nonzero value.

5 The ‘‘invalid’’ and ‘‘divide-by-zero’’ floating-point exceptions are raised as specified in subsequent subclauses of this annex.

6 The ‘‘overflow’’ floating-point exception is raised whenever an infinity — or, because of rounding direction, a maximal-magnitude finite number — is returned in lieu of a value whose magnitude is too large.

7 The ‘‘underflow’’ floating-point exception is raised whenever a result is tiny (essentially subnormal or zero) and suffers loss of accuracy.[371]

---

370) $0 - 0$ yields -0 instead of $+0$ just when the rounding direction is downward.

371) IEC 60559 allows different definitions of underflow. They all result in the same values, but differ on when the floating-point exception is raised.

8 Whether or when library functions raise the ‘‘inexact’’ floating-point exception is unspecified, unless explicitly specified otherwise.

9 Whether or when library functions raise an undeserved ‘‘underflow’’ floating-point exception is unspecified.[372)] Otherwise, as implied by F.8.6, the **`<math.h>`** functions do not raise spurious floating-point exceptions (detectable by the user), other than the ‘‘inexact’’ floating-point exception.

10 Whether the functions honor the rounding direction mode is implementation-defined, unless explicitly specified otherwise.

11 Functions with a NaN argument return a NaN result and raise no floating-point exception, except where stated otherwise.

12 The specifications in the following subclauses append to the definitions in **`<math.h>`**. For families of functions, the specifications apply to all of the functions even though only the principal function is shown. Unless otherwise specified, where the symbol ‘‘±’’ occurs in both an argument and the result, the result has the same sign as the argument.

Recommended practice

13 If a function with one or more NaN arguments returns a NaN result, the result should be the same as one of the NaN arguments (after possible type conversion), except perhaps for the sign.

F.10.1 Trigonometric functions

F.10.1.1 The `acos` functions

1 — **`acos(`**1**`)`** returns +0.

— **`acos(`***x***`)`** returns a NaN and raises the ‘‘invalid’’ floating-point exception for $|x| > 1$.

F.10.1.2 The `asin` functions

1 — **`asin(`**±0**`)`** returns ±0.

— **`asin(`***x***`)`** returns a NaN and raises the ‘‘invalid’’ floating-point exception for $|x| > 1$.

372) It is intended that undeserved ‘‘underflow’’ and ‘‘inexact’’ floating-point exceptions are raised only if avoiding them would be too costly.

F.10.1.3 The `atan` functions

1 — **`atan`**`(`± 0`)` returns ± 0.

— **`atan`**`(`$\pm\infty$`)` returns $\pm\pi/2$.

F.10.1.4 The `atan2` functions

1 — **`atan2`**`(`± 0`, `-0`)` returns $\pm\pi$.[373)]

— **`atan2`**`(`± 0`, `$+0$`)` returns ± 0.

— **`atan2`**`(`± 0`, `x`)` returns $\pm\pi$ for $x < 0$.

— **`atan2`**`(`± 0`, `x`)` returns ± 0 for $x > 0$.

— **`atan2`**`(`y`, `± 0`)` returns $-\pi/2$ for $y < 0$.

— **`atan2`**`(`y`, `± 0`)` returns $\pi/2$ for $y > 0$.

— **`atan2`**`(`$\pm y$`, `$-\infty$`)` returns $\pm\pi$ for finite $y > 0$.

— **`atan2`**`(`$\pm y$`, `$+\infty$`)` returns ± 0 for finite $y > 0$.

— **`atan2`**`(`$\pm\infty$`, `x`)` returns $\pm\pi/2$ for finite x.

— **`atan2`**`(`$\pm\infty$`, `$-\infty$`)` returns $\pm 3\pi/4$.

— **`atan2`**`(`$\pm\infty$`, `$+\infty$`)` returns $\pm\pi/4$.

F.10.1.5 The `cos` functions

1 — **`cos`**`(`± 0`)` returns 1.

— **`cos`**`(`$\pm\infty$`)` returns a NaN and raises the ‘‘invalid’’ floating-point exception.

F.10.1.6 The `sin` functions

1 — **`sin`**`(`± 0`)` returns ± 0.

— **`sin`**`(`$\pm\infty$`)` returns a NaN and raises the ‘‘invalid’’ floating-point exception.

F.10.1.7 The `tan` functions

1 — **`tan`**`(`± 0`)` returns ± 0.

— **`tan`**`(`$\pm\infty$`)` returns a NaN and raises the ‘‘invalid’’ floating-point exception.

---

373) **`atan2`**`(0, 0)` does not raise the ‘‘invalid’’ floating-point exception, nor does **`atan2`**`(`y`, 0)` raise the ‘‘divide-by-zero’’ floating-point exception.

F.10.2 Hyperbolic functions

F.10.2.1 The `acosh` functions

1 — **`acosh(1)`** returns +0.

— **`acosh(`*x*`)`** returns a NaN and raises the ‘‘invalid’’ floating-point exception for $x < 1$.

— **`acosh(+∞)`** returns +∞.

F.10.2.2 The `asinh` functions

1 — **`asinh(±0)`** returns ±0.

— **`asinh(±∞)`** returns ±∞.

F.10.2.3 The `atanh` functions

1 — **`atanh(±0)`** returns ±0.

— **`atanh(±1)`** returns ±∞ and raises the ‘‘divide-by-zero’’ floating-point exception.

— **`atanh(`*x*`)`** returns a NaN and raises the ‘‘invalid’’ floating-point exception for $|x| > 1$.

F.10.2.4 The `cosh` functions

1 — **`cosh(±0)`** returns 1.

— **`cosh(±∞)`** returns +∞.

F.10.2.5 The `sinh` functions

1 — **`sinh(±0)`** returns ±0.

— **`sinh(±∞)`** returns ±∞.

F.10.2.6 The `tanh` functions

1 — **`tanh(±0)`** returns ±0.

— **`tanh(±∞)`** returns ±1.

F.10.3 Exponential and logarithmic functions

F.10.3.1 The `exp` functions

1 — **`exp(±0)`** returns 1.

— **`exp(−∞)`** returns +0.

— **`exp(+∞)`** returns +∞.

F.10.3.2 The `exp2` functions

1 — `exp2(±0)` returns 1.

— `exp2(−∞)` returns +0.

— `exp2(+∞)` returns +∞.

F.10.3.3 The `expm1` functions

1 — `expm1(±0)` returns ±0.

— `expm1(−∞)` returns −1.

— `expm1(+∞)` returns +∞.

F.10.3.4 The `frexp` functions

1 — `frexp(±0, exp)` returns ±0, and stores 0 in the object pointed to by **`exp`**.

— `frexp(±∞, exp)` returns ±∞, and stores an unspecified value in the object pointed to by **`exp`**.

— `frexp(NaN, exp)` stores an unspecified value in the object pointed to by **`exp`** (and returns a NaN).

2 **`frexp`** raises no floating-point exceptions.

3 When the radix of the argument is a power of 2, the returned value is exact and is independent of the current rounding direction mode.

4 On a binary system, the body of the **`frexp`** function might be

```
{
      *exp = (value == 0) ? 0 : (int)(1 + logb(value));
      return scalbn(value, -(*exp));
}
```

F.10.3.5 The `ilogb` functions

1 When the correct result is representable in the range of the return type, the returned value is exact and is independent of the current rounding direction mode.

2 If the correct result is outside the range of the return type, the numeric result is unspecified and the ‘‘invalid’’ floating-point exception is raised.

3 `ilogb(`*x*`)`, for *x* zero, infinite, or NaN, raises the ‘‘invalid’’ floating-point exception and returns the value specified in 7.12.6.5.

F.10.3.6 The `ldexp` functions

1 On a binary system, **`ldexp(x, exp)`** is equivalent to **`scalbn(x, exp)`**.

F.10.3.7 The `log` functions

1 — **`log(`**± 0**`)`** returns $-\infty$ and raises the ''divide-by-zero'' floating-point exception.

— **`log(`**1**`)`** returns +0.

— **`log(`***x***`)`** returns a NaN and raises the ''invalid'' floating-point exception for $x < 0$.

— **`log(`**$+\infty$**`)`** returns $+\infty$.

F.10.3.8 The `log10` functions

1 — **`log10(`**± 0**`)`** returns $-\infty$ and raises the ''divide-by-zero'' floating-point exception.

— **`log10(`**1**`)`** returns +0.

— **`log10(`***x***`)`** returns a NaN and raises the ''invalid'' floating-point exception for $x < 0$.

— **`log10(`**$+\infty$**`)`** returns $+\infty$.

F.10.3.9 The `log1p` functions

1 — **`log1p(`**± 0**`)`** returns ± 0.

— **`log1p(`**-1**`)`** returns $-\infty$ and raises the ''divide-by-zero'' floating-point exception.

— **`log1p(`***x***`)`** returns a NaN and raises the ''invalid'' floating-point exception for $x < -1$.

— **`log1p(`**$+\infty$**`)`** returns $+\infty$.

F.10.3.10 The `log2` functions

1 — **`log2(`**± 0**`)`** returns $-\infty$ and raises the ''divide-by-zero'' floating-point exception.

— **`log2(`**1**`)`** returns +0.

— **`log2(`***x***`)`** returns a NaN and raises the ''invalid'' floating-point exception for $x < 0$.

— **`log2(`**$+\infty$**`)`** returns $+\infty$.

F.10.3.11 The `logb` functions

1 — **`logb(`**± 0**`)`** returns $-\infty$ and raises the ''divide-by-zero'' floating-point exception.

— **`logb(`**$\pm\infty$**`)`** returns $+\infty$.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.3.12 The `modf` functions

1 — **`modf(`**±*x*, **`iptr)`** returns a result with the same sign as *x*.

— **`modf(`**±∞, **`iptr)`** returns ±0 and stores ±∞ in the object pointed to by **`iptr`**.

— **`modf(`**NaN, **`iptr)`** stores a NaN in the object pointed to by **`iptr`** (and returns a NaN).

2 The returned values are exact and are independent of the current rounding direction mode.

3 **`modf`** behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double modf(double value, double *iptr)
{
     int save_round = fegetround();
     fesetround(FE_TOWARDZERO);
     *iptr = nearbyint(value);
     fesetround(save_round);
     return copysign(
          isinf(value) ? 0.0 :
               value - (*iptr), value);
}
```

F.10.3.13 The `scalbn` and `scalbln` functions

1 — **`scalbn(`**±0, *n***`)`** returns ±0.

— **`scalbn(`***x*, 0**`)`** returns *x*.

— **`scalbn(`**±∞, *n***`)`** returns ±∞.

2 If the calculation does not overflow or underflow, the returned value is exact and independent of the current rounding direction mode.

F.10.4 Power and absolute value functions

F.10.4.1 The `cbrt` functions

1 — `cbrt(`±0`)` returns ±0.

— `cbrt(`±∞`)` returns ±∞.

F.10.4.2 The `fabs` functions

1 — `fabs(`±0`)` returns +0.

— `fabs(`±∞`)` returns +∞.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.4.3 The `hypot` functions

1 — `hypot(`*x*`,` *y*`)`, `hypot(`*y*`,` *x*`)`, and `hypot(`*x*`,` −*y*`)` are equivalent.

— `hypot(`*x*`,` ±0`)` is equivalent to `fabs(`*x*`)`.

— `hypot(`±∞`,` *y*`)` returns +∞, even if *y* is a NaN.

F.10.4.4 The `pow` functions

1 — `pow(`±0`,` *y*`)` returns ±∞ and raises the ‘‘divide-by-zero’’ floating-point exception for *y* an odd integer < 0.

— `pow(`±0`,` *y*`)` returns +∞ and raises the ‘‘divide-by-zero’’ floating-point exception for *y* < 0, finite, and not an odd integer.

— `pow(`±0`,` −∞`)` returns +∞ and may raise the ‘‘divide-by-zero’’ floating-point exception.

— `pow(`±0`,` *y*`)` returns ±0 for *y* an odd integer > 0.

— `pow(`±0`,` *y*`)` returns +0 for *y* > 0 and not an odd integer.

— `pow(`−1`,` ±∞`)` returns 1.

— `pow(`+1`,` *y*`)` returns 1 for any *y*, even a NaN.

— `pow(`*x*`,` ±0`)` returns 1 for any *x*, even a NaN.

— `pow(`*x*`,` *y*`)` returns a NaN and raises the ‘‘invalid’’ floating-point exception for finite *x* < 0 and finite non-integer *y*.

— `pow(`*x*`,` −∞`)` returns +∞ for | *x* | < 1.

— `pow(`*x*`,` −∞`)` returns +0 for | *x* | > 1.

— `pow(`*x*`,` +∞`)` returns +0 for | *x* | < 1.

— `pow(`*x*`,` +∞`)` returns +∞ for | *x* | > 1.

— **`pow`**`(`$-\infty$`,` *y*`)` returns -0 for *y* an odd integer < 0.

— **`pow`**`(`$-\infty$`,` *y*`)` returns +0 for *y* < 0 and not an odd integer.

— **`pow`**`(`$-\infty$`,` *y*`)` returns $-\infty$ for *y* an odd integer > 0.

— **`pow`**`(`$-\infty$`,` *y*`)` returns $+\infty$ for *y* > 0 and not an odd integer.

— **`pow`**`(`$+\infty$`,` *y*`)` returns +0 for *y* < 0.

— **`pow`**`(`$+\infty$`,` *y*`)` returns $+\infty$ for *y* > 0.

F.10.4.5 The `sqrt` functions

1 **`sqrt`** is fully specified as a basic arithmetic operation in IEC 60559. The returned value is dependent on the current rounding direction mode.

F.10.5 Error and gamma functions

F.10.5.1 The `erf` functions

1 — **`erf`**`(`±0`)` returns ±0.

— **`erf`**`(`$\pm\infty$`)` returns ±1.

F.10.5.2 The `erfc` functions

1 — **`erfc`**`(`$-\infty$`)` returns 2.

— **`erfc`**`(`$+\infty$`)` returns +0.

F.10.5.3 The `lgamma` functions

1 — **`lgamma`**`(1)` returns +0.

— **`lgamma`**`(2)` returns +0.

— **`lgamma`**`(`*x*`)` returns $+\infty$ and raises the ‘‘divide-by-zero’’ floating-point exception for *x* a negative integer or zero.

— **`lgamma`**`(`$-\infty$`)` returns $+\infty$.

— **`lgamma`**`(`$+\infty$`)` returns $+\infty$.

F.10.5.4 The `tgamma` functions

1 — **`tgamma`**`(`±0`)` returns $\pm\infty$ and raises the ‘‘divide-by-zero’’ floating-point exception.

— **`tgamma`**`(`*x*`)` returns a NaN and raises the ‘‘invalid’’ floating-point exception for *x* a negative integer.

— **`tgamma`**`(`$-\infty$`)` returns a NaN and raises the ‘‘invalid’’ floating-point exception.

— **`tgamma`**`(`$+\infty$`)` returns $+\infty$.

F.10.6 Nearest integer functions

F.10.6.1 The `ceil` functions

1 — `ceil(`±0`)` returns ±0.

— `ceil(`±∞`)` returns ±∞.

2 The returned value is independent of the current rounding direction mode.

3 The `double` version of `ceil` behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double ceil(double x)
{
     double result;
     int save_round = fegetround();
     fesetround(FE_UPWARD);
     result = rint(x); // or nearbyint instead of rint
     fesetround(save_round);
     return result;
}
```

4 The `ceil` functions may, but are not required to, raise the ‘‘inexact’’ floating-point exception for finite non-integer arguments, as this implementation does.

F.10.6.2 The `floor` functions

1 — `floor(`±0`)` returns ±0.

— `floor(`±∞`)` returns ±∞.

2 The returned value and is independent of the current rounding direction mode.

3 See the sample implementation for `ceil` in F.10.6.1. The `floor` functions may, but are not required to, raise the ‘‘inexact’’ floating-point exception for finite non-integer arguments, as that implementation does.

F.10.6.3 The `nearbyint` functions

1 The `nearbyint` functions use IEC 60559 rounding according to the current rounding direction. They do not raise the ‘‘inexact’’ floating-point exception if the result differs in value from the argument.

— `nearbyint(`±0`)` returns ±0 (for all rounding directions).

— `nearbyint(`±∞`)` returns ±∞ (for all rounding directions).

F.10.6.4 The `rint` functions

1 The **`rint`** functions differ from the **`nearbyint`** functions only in that they do raise the ‘‘inexact’’ floating-point exception if the result differs in value from the argument.

F.10.6.5 The `lrint` and `llrint` functions

1 The **`lrint`** and **`llrint`** functions provide floating-to-integer conversion as prescribed by IEC 60559. They round according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and the ‘‘invalid’’ floating-point exception is raised. When they raise no other floating-point exception and the result differs from the argument, they raise the ‘‘inexact’’ floating-point exception.

F.10.6.6 The `round` functions

1 — **`round(`**± 0**`)`** returns ± 0.

— **`round(`**$\pm\infty$**`)`** returns $\pm\infty$.

2 The returned value is independent of the current rounding direction mode.

3 The **`double`** version of **`round`** behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double round(double x)
{
     double result;
     fenv_t save_env;
     feholdexcept(&save_env);
     result = rint(x);
     if (fetestexcept(FE_INEXACT)) {
          fesetround(FE_TOWARDZERO);
          result = rint(copysign(0.5 + fabs(x), x));
     }
     feupdateenv(&save_env);
     return result;
}
```

The **`round`** functions may, but are not required to, raise the ‘‘inexact’’ floating-point exception for finite non-integer numeric arguments, as this implementation does.

F.10.6.7 The `lround` and `llround` functions

1 The **`lround`** and **`llround`** functions differ from the **`lrint`** and **`llrint`** functions with the default rounding direction just in that the **`lround`** and **`llround`** functions round halfway cases away from zero and need not raise the ‘‘inexact’’ floating-point exception for non-integer arguments that round to within the range of the return type.

F.10.6.8 The `trunc` functions

1 The **`trunc`** functions use IEC 60559 rounding toward zero (regardless of the current rounding direction). The returned value is exact.

— **`trunc(`**±0**`)`** returns ±0.

— **`trunc(`**±∞**`)`** returns ±∞.

2 The returned value is independent of the current rounding direction mode. The **`trunc`** functions may, but are not required to, raise the ‘‘inexact’’ floating-point exception for finite non-integer arguments.

F.10.7 Remainder functions

F.10.7.1 The `fmod` functions

1 — **`fmod(`**±0**`,`** *y***`)`** returns ±0 for *y* not zero.

— **`fmod(`***x***`,`** *y***`)`** returns a NaN and raises the ‘‘invalid’’ floating-point exception for *x* infinite or *y* zero (and neither is a NaN).

— **`fmod(`***x***`,`** ±∞**`)`** returns *x* for *x* not infinite.

2 When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

3 The **`double`** version of **`fmod`** behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double fmod(double x, double y)
{
      double result;
      result = remainder(fabs(x), (y = fabs(y)));
      if (signbit(result)) result += y;
      return copysign(result, x);
}
```

F.10.7.2 The `remainder` functions

1 The **`remainder`** functions are fully specified as a basic arithmetic operation in IEC 60559.

2 When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

F.10.7.3 The `remquo` functions

1 The **`remquo`** functions follow the specifications for the **`remainder`** functions. They have no further specifications special to IEC 60559 implementations.

2 When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

F.10.8 Manipulation functions

F.10.8.1 The `copysign` functions

1 **`copysign`** is specified in the Appendix to IEC 60559.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.8.2 The `nan` functions

1 All IEC 60559 implementations support quiet NaNs, in all floating formats.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.8.3 The `nextafter` functions

1 — **`nextafter(`***x*, *y***`)`** raises the ''overflow'' and ''inexact'' floating-point exceptions for *x* finite and the function value infinite.

— **`nextafter(`***x*, *y***`)`** raises the ''underflow'' and ''inexact'' floating-point exceptions for the function value subnormal or zero and $x \neq y$.

2 Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

F.10.8.4 The `nexttoward` functions

1 No additional requirements beyond those on **`nextafter`**.

2 Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

F.10.9 Maximum, minimum, and positive difference functions

F.10.9.1 The `fdim` functions

1 No additional requirements.

F.10.9.2 The `fmax` functions

1 If just one argument is a NaN, the **`fmax`** functions return the other argument (if both arguments are NaNs, the functions return a NaN).

2 The returned value is exact and is independent of the current rounding direction mode.

3 The body of the **`fmax`** function might be[374)]

```
{ return (isgreaterequal(x, y) ||
      isnan(y)) ? x : y; }
```

F.10.9.3 The `fmin` functions

1 The **`fmin`** functions are analogous to the **`fmax`** functions (see F.10.9.2).

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.10 Floating multiply-add

F.10.10.1 The `fma` functions

1 — **`fma(`***x*, *y*, *z***`)`** computes $xy + z$, correctly rounded once.

— **`fma(`***x*, *y*, *z***`)`** returns a NaN and optionally raises the ‘‘invalid’’ floating-point exception if one of *x* and *y* is infinite, the other is zero, and *z* is a NaN.

— **`fma(`***x*, *y*, *z***`)`** returns a NaN and raises the ‘‘invalid’’ floating-point exception if one of *x* and *y* is infinite, the other is zero, and *z* is not a NaN.

— **`fma(`***x*, *y*, *z***`)`** returns a NaN and raises the ‘‘invalid’’ floating-point exception if *x* times *y* is an exact infinity and *z* is also an infinity but with the opposite sign.

374) Ideally, **`fmax`** would be sensitive to the sign of zero, for example **`fmax(`**-0.0, $+0.0$**`)`** would return $+0$; however, implementation in software might be impractical.

F.10.11 Comparison macros

1 Relational operators and their corresponding comparison macros (7.12.14) produce equivalent result values, even if argument values are represented in wider formats. Thus, comparison macro arguments represented in formats wider than their semantic types are not converted to the semantic types, unless the wide evaluation method converts operands of relational operators to their semantic types. The standard wide evaluation methods characterized by **`FLT_EVAL_METHOD`** equal to 1 or 2 (5.2.4.2.2), do not convert operands of relational operators to their semantic types.

Annex G

(normative)

IEC 60559-compatible complex arithmetic

G.1 Introduction

1 This annex supplements annex F to specify complex arithmetic for compatibility with IEC 60559 real floating-point arithmetic. An implementation that defines **`__STDC_IEC_559_COMPLEX__`** shall conform to the specifications in this annex.[375)]

G.2 Types

1 There is a new keyword **`_Imaginary`**, which is used to specify imaginary types. It is used as a type specifier within declaration specifiers in the same way as **`_Complex`** is (thus, **`_Imaginary float`** is a valid type name).

2 There are three *imaginary types*, designated as **`float _Imaginary`**, **`double _Imaginary`**, and **`long double _Imaginary`**. The imaginary types (along with the real floating and complex types) are floating types.

3 For imaginary types, the corresponding real type is given by deleting the keyword **`_Imaginary`** from the type name.

4 Each imaginary type has the same representation and alignment requirements as the corresponding real type. The value of an object of imaginary type is the value of the real representation times the imaginary unit.

5 The *imaginary type domain* comprises the imaginary types.

G.3 Conventions

1 A complex or imaginary value with at least one infinite part is regarded as an *infinity* (even if its other part is a NaN). A complex or imaginary value is a *finite number* if each of its parts is a finite number (neither infinite nor NaN). A complex or imaginary value is a *zero* if each of its parts is a zero.

375) Implementations that do not define **`__STDC_IEC_559_COMPLEX__`** are not required to conform to these specifications.

G.4 Conversions

G.4.1 Imaginary types

1 Conversions among imaginary types follow rules analogous to those for real floating types.

G.4.2 Real and imaginary

1 When a value of imaginary type is converted to a real type other than **`_Bool`**,[376)] the result is a positive zero.

2 When a value of real type is converted to an imaginary type, the result is a positive imaginary zero.

G.4.3 Imaginary and complex

1 When a value of imaginary type is converted to a complex type, the real part of the complex result value is a positive zero and the imaginary part of the complex result value is determined by the conversion rules for the corresponding real types.

2 When a value of complex type is converted to an imaginary type, the real part of the complex value is discarded and the value of the imaginary part is converted according to the conversion rules for the corresponding real types.

G.5 Binary operators

1 The following subclauses supplement 6.5 in order to specify the type of the result for an operation with an imaginary operand.

2 For most operand types, the value of the result of a binary operator with an imaginary or complex operand is completely determined, with reference to real arithmetic, by the usual mathematical formula. For some operand types, the usual mathematical formula is problematic because of its treatment of infinities and because of undue overflow or underflow; in these cases the result satisfies certain properties (specified in G.5.1), but is not completely determined.

376) See 6.3.1.2.

G.5.1 Multiplicative operators

Semantics

1 If one operand has real type and the other operand has imaginary type, then the result has imaginary type. If both operands have imaginary type, then the result has real type. (If either operand has complex type, then the result has complex type.)

2 If the operands are not both complex, then the result and floating-point exception behavior of the `*` operator is defined by the usual mathematical formula:

`*`	u	iv	$u + iv$
x	xu	$i(xv)$	$(xu) + i(xv)$
iy	$i(yu)$	$-yv$	$(-yv) + i(yu)$
$x + iy$	$(xu) + i(yu)$	$(-yv) + i(xv)$	

3 If the second operand is not complex, then the result and floating-point exception behavior of the `/` operator is defined by the usual mathematical formula:

`/`	u	iv
x	x/u	$i(-x/v)$
iy	$i(y/u)$	y/v
$x + iy$	$(x/u) + i(y/u)$	$(y/v) + i(-x/v)$

4 The `*` and `/` operators satisfy the following infinity properties for all real, imaginary, and complex operands:[377)]

— if one operand is an infinity and the other operand is a nonzero finite number or an infinity, then the result of the `*` operator is an infinity;

— if the first operand is an infinity and the second operand is a finite number, then the result of the `/` operator is an infinity;

— if the first operand is a finite number and the second operand is an infinity, then the result of the `/` operator is a zero;

377) These properties are already implied for those cases covered in the tables, but are required for all cases (at least where the state for **`CX_LIMITED_RANGE`** is ‘‘off’’).

— if the first operand is a nonzero finite number or an infinity and the second operand is a zero, then the result of the **/** operator is an infinity.

5 If both operands of the ***** operator are complex or if the second operand of the **/** operator is complex, the operator raises floating-point exceptions if appropriate for the calculation of the parts of the result, and may raise spurious floating-point exceptions.

6 EXAMPLE 1 Multiplication of **double _Complex** operands could be implemented as follows. Note that the imaginary unit **I** has imaginary type (see G.6).

```
#include <math.h>
#include <complex.h>

/* Multiply z * w ... */
double complex _Cmultd(double complex z, double complex w)
{
      #pragma STDC FP_CONTRACT OFF
      double a, b, c, d, ac, bd, ad, bc, x, y;
      a = creal(z); b = cimag(z);
      c = creal(w); d = cimag(w);
      ac = a * c;   bd = b * d;
      ad = a * d;   bc = b * c;
      x = ac - bd;  y = ad + bc;
      if (isnan(x) && isnan(y)) {
            /* Recover infinities that computed as NaN+iNaN ... */
            int recalc = 0;
            if (isinf(a) || isinf(b)) { // z is infinite
                  /* "Box" the infinity and change NaNs in the other factor to 0 */
                  a = copysign(isinf(a) ? 1.0 : 0.0, a);
                  b = copysign(isinf(b) ? 1.0 : 0.0, b);
                  if (isnan(c)) c = copysign(0.0, c);
                  if (isnan(d)) d = copysign(0.0, d);
                  recalc = 1;
            }
            if (isinf(c) || isinf(d)) { // w is infinite
                  /* "Box" the infinity and change NaNs in the other factor to 0 */
                  c = copysign(isinf(c) ? 1.0 : 0.0, c);
                  d = copysign(isinf(d) ? 1.0 : 0.0, d);
                  if (isnan(a)) a = copysign(0.0, a);
                  if (isnan(b)) b = copysign(0.0, b);
                  recalc = 1;
            }
            if (!recalc && (isinf(ac) || isinf(bd) ||
                            isinf(ad) || isinf(bc))) {
                  /* Recover infinities from overflow by changing NaNs to 0 ... */
                  if (isnan(a)) a = copysign(0.0, a);
                  if (isnan(b)) b = copysign(0.0, b);
                  if (isnan(c)) c = copysign(0.0, c);
                  if (isnan(d)) d = copysign(0.0, d);
                  recalc = 1;
            }
            if (recalc) {
```

```
                        x = INFINITY * ( a * c - b * d );
                        y = INFINITY * ( a * d + b * c );
                }
        }
        return x + I * y;
}
```

7 This implementation achieves the required treatment of infinities at the cost of only one **isnan** test in ordinary (finite) cases. It is less than ideal in that undue overflow and underflow may occur.

8 EXAMPLE 2 Division of two **double _Complex** operands could be implemented as follows.

```
#include <math.h>
#include <complex.h>

/* Divide z / w ... */
double complex _Cdivd(double complex z, double complex w)
{
        #pragma STDC FP_CONTRACT OFF
        double a, b, c, d, logbw, denom, x, y;
        int ilogbw = 0;
        a = creal(z); b = cimag(z);
        c = creal(w); d = cimag(w);
        logbw = logb(fmax(fabs(c), fabs(d)));
        if (isfinite(logbw)) {
                ilogbw = (int)logbw;
                c = scalbn(c, -ilogbw); d = scalbn(d, -ilogbw);
        }
        denom = c * c + d * d;
        x = scalbn((a * c + b * d) / denom, -ilogbw);
        y = scalbn((b * c - a * d) / denom, -ilogbw);

        /* Recover infinities and zeros that computed as NaN+iNaN;               */
        /* the only cases are nonzero/zero, infinite/finite, and finite/infinite, ... */

        if (isnan(x) && isnan(y)) {
                if ((denom == 0.0) &&
                        (!isnan(a) || !isnan(b))) {
                        x = copysign(INFINITY, c) * a;
                        y = copysign(INFINITY, c) * b;
                }
                else if ((isinf(a) || isinf(b)) &&
                        isfinite(c) && isfinite(d)) {
                        a = copysign(isinf(a) ? 1.0 : 0.0, a);
                        b = copysign(isinf(b) ? 1.0 : 0.0, b);
                        x = INFINITY * ( a * c + b * d );
                        y = INFINITY * ( b * c - a * d );
                }
                else if ((logbw == INFINITY) &&
                        isfinite(a) && isfinite(b)) {
                        c = copysign(isinf(c) ? 1.0 : 0.0, c);
                        d = copysign(isinf(d) ? 1.0 : 0.0, d);
                        x = 0.0 * ( a * c + b * d );
                        y = 0.0 * ( b * c - a * d );
```

```
                }
            }
            return x + I * y;
    }
```

9 Scaling the denominator alleviates the main overflow and underflow problem, which is more serious than for multiplication. In the spirit of the multiplication example above, this code does not defend against overflow and underflow in the calculation of the numerator. Scaling with the **scalbn** function, instead of with division, provides better roundoff characteristics.

G.5.2 Additive operators

Semantics

1 If both operands have imaginary type, then the result has imaginary type. (If one operand has real type and the other operand has imaginary type, or if either operand has complex type, then the result has complex type.)

2 In all cases the result and floating-point exception behavior of a **+** or **-** operator is defined by the usual mathematical formula:

+ or **-**	u	iv	$u + iv$
x	$x \pm u$	$x \pm iv$	$(x \pm u) \pm iv$
iy	$\pm u + iy$	$i(y \pm v)$	$\pm u + i(y \pm v)$
$x + iy$	$(x \pm u) + iy$	$x + i(y \pm v)$	$(x \pm u) + i(y \pm v)$

G.6 Complex arithmetic <complex.h>

1 The macros

```
imaginary
```

and

```
_Imaginary_I
```

are defined, respectively, as **_Imaginary** and a constant expression of type **const float _Imaginary** with the value of the imaginary unit. The macro

```
I
```

is defined to be **_Imaginary_I** (not **_Complex_I** as stated in 7.3). Notwithstanding the provisions of 7.1.3, a program may undefine and then perhaps redefine the macro **imaginary**.

2 This subclause contains specifications for the **<complex.h>** functions that are particularly suited to IEC 60559 implementations. For families of functions, the specifications apply to all of the functions even though only the principal function is

shown. Unless otherwise specified, where the symbol ‘‘±’’ occurs in both an argument and the result, the result has the same sign as the argument.

3 The functions are continuous onto both sides of their branch cuts, taking into account the sign of zero. For example, **csqrt(**$-2 \pm i0$**)** $= \pm i\sqrt{2}$.

4 Since complex and imaginary values are composed of real values, each function may be regarded as computing real values from real values. Except as noted, the functions treat real infinities, NaNs, signed zeros, subnormals, and the floating-point exception flags in a manner consistent with the specifications for real functions in F.10.[378)]

5 The functions **cimag**, **conj**, **cproj**, and **creal** are fully specified for all implementations, including IEC 60559 ones, in 7.3.9. These functions raise no floating-point exceptions.

6 Each of the functions **cabs** and **carg** is specified by a formula in terms of a real function (whose special cases are covered in annex F):

cabs($x + iy$**)** = **hypot(**x**,** y**)**
carg($x + iy$**)** = **atan2(**y**,** x**)**

7 Each of the functions **casin**, **catan**, **ccos**, **csin**, and **ctan** is specified implicitly by a formula in terms of other complex functions (whose special cases are specified below):

casin(z**)** = $-i$ **casinh(**iz**)**
catan(z**)** = $-i$ **catanh(**iz**)**
ccos(z**)** = **ccosh(**iz**)**
csin(z**)** = $-i$ **csinh(**iz**)**
ctan(z**)** = $-i$ **ctanh(**iz**)**

8 For the other functions, the following subclauses specify behavior for special cases, including treatment of the ‘‘invalid’’ and ‘‘divide-by-zero’’ floating-point exceptions. For families of functions, the specifications apply to all of the functions even though only the principal function is shown. For a function f satisfying $f(\mathrm{conj}(z)) = \mathrm{conj}(f(z))$, the specifications for the upper half-plane imply the specifications for the lower half-plane; if the function f is also either even, $f(-z) = f(z)$, or odd, $f(-z) = -f(z)$, then the specifications for the first quadrant imply the specifications for the other three quadrants.

9 In the following subclauses, $\mathrm{cis}(y)$ is defined as $\cos(y) + i\sin(y)$.

378) As noted in G.3, a complex value with at least one infinite part is regarded as an infinity even if its other part is a NaN.

G.6.1 Trigonometric functions

G.6.1.1 The `cacos` functions

1 — `cacos(conj(`z`))` = `conj(cacos(`z`))`.

— `cacos(`$\pm 0 + i0$`)` returns $\pi/2 - i0$.

— `cacos(`$\pm 0 + i\mathrm{NaN}$`)` returns $\pi/2 + i\mathrm{NaN}$.

— `cacos(`$x + i\infty$`)` returns $\pi/2 - i\infty$, for finite x.

— `cacos(`$x + i\mathrm{NaN}$`)` returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the ‘‘invalid’’ floating-point exception, for nonzero finite x.

— `cacos(`$-\infty + iy$`)` returns $\pi - i\infty$, for positive-signed finite y.

— `cacos(`$+\infty + iy$`)` returns $+0 - i\infty$, for positive-signed finite y.

— `cacos(`$-\infty + i\infty$`)` returns $3\pi/4 - i\infty$.

— `cacos(`$+\infty + i\infty$`)` returns $\pi/4 - i\infty$.

— `cacos(`$\pm\infty + i\mathrm{NaN}$`)` returns $\mathrm{NaN} \pm i\infty$ (where the sign of the imaginary part of the result is unspecified).

— `cacos(`$\mathrm{NaN} + iy$`)` returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the ‘‘invalid’’ floating-point exception, for finite y.

— `cacos(`$\mathrm{NaN} + i\infty$`)` returns $\mathrm{NaN} - i\infty$.

— `cacos(`$\mathrm{NaN} + i\mathrm{NaN}$`)` returns $\mathrm{NaN} + i\mathrm{NaN}$.

G.6.2 Hyperbolic functions

G.6.2.1 The `cacosh` functions

1 — `cacosh(conj(`z`))` = `conj(cacosh(`z`))`.

— `cacosh(`$\pm 0 + i0$`)` returns $+0 + i\pi/2$.

— `cacosh(`$x + i\infty$`)` returns $+\infty + i\pi/2$, for finite x.

— `cacosh(`$x + i\mathrm{NaN}$`)` returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the ‘‘invalid’’ floating-point exception, for finite x.

— `cacosh(`$-\infty + iy$`)` returns $+\infty + i\pi$, for positive-signed finite y.

— `cacosh(`$+\infty + iy$`)` returns $+\infty + i0$, for positive-signed finite y.

— `cacosh(`$-\infty + i\infty$`)` returns $+\infty + i3\pi/4$.

— `cacosh(`$+\infty + i\infty$`)` returns $+\infty + i\pi/4$.

— `cacosh(`$\pm\infty + i\mathrm{NaN}$`)` returns $+\infty + i\mathrm{NaN}$.

— **`cacosh`**`(`NaN + iy`)` returns NaN + iNaN and optionally raises the ‘‘invalid’’ floating-point exception, for finite y.

— **`cacosh`**`(`NaN + $i\infty$`)` returns +∞ + iNaN.

— **`cacosh`**`(`NaN + iNaN`)` returns NaN + iNaN.

G.6.2.2 The `casinh` functions

1 — **`casinh(conj(`*z*`)) = conj(casinh(`*z*`))`** and **`casinh`** is odd.

— **`casinh`**`(`+0 + i0`)` returns 0 + i0.

— **`casinh`**`(`$x + i\infty$`)` returns +∞ + $i\pi/2$ for positive-signed finite x.

— **`casinh`**`(`x + iNaN`)` returns NaN + iNaN and optionally raises the ‘‘invalid’’ floating-point exception, for finite x.

— **`casinh`**`(`+∞ + iy`)` returns +∞ + i0 for positive-signed finite y.

— **`casinh`**`(`+∞ + $i\infty$`)` returns +∞ + $i\pi/4$.

— **`casinh`**`(`+∞ + iNaN`)` returns +∞ + iNaN.

— **`casinh`**`(`NaN + i0`)` returns NaN + i0.

— **`casinh`**`(`NaN + iy`)` returns NaN + iNaN and optionally raises the ‘‘invalid’’ floating-point exception, for finite nonzero y.

— **`casinh`**`(`NaN + $i\infty$`)` returns ±∞ + iNaN (where the sign of the real part of the result is unspecified).

— **`casinh`**`(`NaN + iNaN`)` returns NaN + iNaN.

G.6.2.3 The `catanh` functions

1 — **`catanh(conj(`*z*`)) = conj(catanh(`*z*`))`** and **`catanh`** is odd.

— **`catanh`**`(`+0 + i0`)` returns +0 + i0.

— **`catanh`**`(`+0 + iNaN`)` returns +0 + iNaN.

— **`catanh`**`(`+1 + i0`)` returns +∞ + i0 and raises the ‘‘divide-by-zero’’ floating-point exception.

— **`catanh`**`(`$x + i\infty$`)` returns +0 + $i\pi/2$, for finite positive-signed x.

— **`catanh`**`(`x + iNaN`)` returns NaN + iNaN and optionally raises the ‘‘invalid’’ floating-point exception, for nonzero finite x.

— **`catanh`**`(`+∞ + iy`)` returns +0 + $i\pi/2$, for finite positive-signed y.

— **`catanh`**`(`+∞ + $i\infty$`)` returns +0 + $i\pi/2$.

— **`catanh`**`(`+∞ + iNaN`)` returns +0 + iNaN.

— **`catanh`**`(`NaN + iy`)` returns NaN + iNaN and optionally raises the ‘‘invalid’’ floating-point exception, for finite y.

— **`catanh`**`(`NaN + $i\infty$`)` returns $\pm 0 + i\pi/2$ (where the sign of the real part of the result is unspecified).

— **`catanh`**`(`NaN + iNaN`)` returns NaN + iNaN.

G.6.2.4 The `ccosh` functions

1 — **`ccosh`**`(`**`conj`**`(`z`))` = **`conj`**`(`**`ccosh`**`(`z`))` and **`ccosh`** is even.

— **`ccosh`**`(`$+0 + i0$`)` returns $1 + i0$.

— **`ccosh`**`(`$+0 + i\infty$`)` returns NaN $\pm\, i0$ (where the sign of the imaginary part of the result is unspecified) and raises the ‘‘invalid’’ floating-point exception.

— **`ccosh`**`(`$+0 + i$NaN`)` returns NaN $\pm\, i0$ (where the sign of the imaginary part of the result is unspecified).

— **`ccosh`**`(`$x + i\infty$`)` returns NaN + iNaN and raises the ‘‘invalid’’ floating-point exception, for finite nonzero x.

— **`ccosh`**`(`$x + i$NaN`)` returns NaN + iNaN and optionally raises the ‘‘invalid’’ floating-point exception, for finite nonzero x.

— **`ccosh`**`(`$+\infty + i0$`)` returns $+\infty + i0$.

— **`ccosh`**`(`$+\infty + iy$`)` returns $+\infty$ cis(y), for finite nonzero y.

— **`ccosh`**`(`$+\infty + i\infty$`)` returns $\pm\infty + i$NaN (where the sign of the real part of the result is unspecified) and raises the ‘‘invalid’’ floating-point exception.

— **`ccosh`**`(`$+\infty + i$NaN`)` returns $+\infty + i$NaN.

— **`ccosh`**`(`NaN + $i0$`)` returns NaN $\pm\, i0$ (where the sign of the imaginary part of the result is unspecified).

— **`ccosh`**`(`NaN + iy`)` returns NaN + iNaN and optionally raises the ‘‘invalid’’ floating-point exception, for all nonzero numbers y.

— **`ccosh`**`(`NaN + iNaN`)` returns NaN + iNaN.

G.6.2.5 The `csinh` functions

1 — **`csinh`**`(`**`conj`**`(`z`))` = **`conj`**`(`**`csinh`**`(`z`))` and **`csinh`** is odd.

— **`csinh`**`(`$+0 + i0$`)` returns $+0 + i0$.

— **`csinh`**`(`$+0 + i\infty$`)` returns $\pm 0 + i$NaN (where the sign of the real part of the result is unspecified) and raises the ‘‘invalid’’ floating-point exception.

— **`csinh`**`(`$+0 + i$NaN`)` returns $\pm 0 + i$NaN (where the sign of the real part of the result is unspecified).

— **`csinh(`**$x + i\infty$**`)`** returns NaN + iNaN and raises the ''invalid'' floating-point exception, for positive finite x.

— **`csinh(`**$x + i$NaN**`)`** returns NaN + iNaN and optionally raises the ''invalid'' floating-point exception, for finite nonzero x.

— **`csinh(`**$+\infty + i0$**`)`** returns $+\infty + i0$.

— **`csinh(`**$+\infty + iy$**`)`** returns $+\infty$ cis(y), for positive finite y.

— **`csinh(`**$+\infty + i\infty$**`)`** returns $\pm\infty + i$NaN (where the sign of the real part of the result is unspecified) and raises the ''invalid'' floating-point exception.

— **`csinh(`**$+\infty + i$NaN**`)`** returns $\pm\infty + i$NaN (where the sign of the real part of the result is unspecified).

— **`csinh(`**NaN $+ i0$**`)`** returns NaN $+ i0$.

— **`csinh(`**NaN $+ iy$**`)`** returns NaN + iNaN and optionally raises the ''invalid'' floating-point exception, for all nonzero numbers y.

— **`csinh(`**NaN + iNaN**`)`** returns NaN + iNaN.

G.6.2.6 The `ctanh` functions

1 — **`ctanh(conj(`**z**`)) = conj(ctanh(`**z**`))`** and **`ctanh`** is odd.

— **`ctanh(`**$+0 + i0$**`)`** returns $+0 + i0$.

— **`ctanh(`**$x + i\infty$**`)`** returns NaN + iNaN and raises the ''invalid'' floating-point exception, for finite x.

— **`ctanh(`**$x + i$NaN**`)`** returns NaN + iNaN and optionally raises the ''invalid'' floating-point exception, for finite x.

— **`ctanh(`**$+\infty + iy$**`)`** returns $1 + i0\sin(2y)$, for positive-signed finite y.

— **`ctanh(`**$+\infty + i\infty$**`)`** returns $1 \pm i0$ (where the sign of the imaginary part of the result is unspecified).

— **`ctanh(`**$+\infty + i$NaN**`)`** returns $1 \pm i0$ (where the sign of the imaginary part of the result is unspecified).

— **`ctanh(`**NaN $+ i0$**`)`** returns NaN $+ i0$.

— **`ctanh(`**NaN $+ iy$**`)`** returns NaN + iNaN and optionally raises the ''invalid'' floating-point exception, for all nonzero numbers y.

— **`ctanh(`**NaN + iNaN**`)`** returns NaN + iNaN.

G.6.3 Exponential and logarithmic functions

G.6.3.1 The `cexp` functions

1 — `cexp(conj(`z`))` = `conj(cexp(`z`))`.

— `cexp(`$\pm 0 + i0$`)` returns $1 + i0$.

— `cexp(`$x + i\infty$`)` returns NaN $+ i$NaN and raises the ‘‘invalid’’ floating-point exception, for finite x.

— `cexp(`$x + i$NaN`)` returns NaN $+ i$NaN and optionally raises the ‘‘invalid’’ floating-point exception, for finite x.

— `cexp(`$+\infty + i0$`)` returns $+\infty + i0$.

— `cexp(`$-\infty + iy$`)` returns $+0$ cis(y), for finite y.

— `cexp(`$+\infty + iy$`)` returns $+\infty$ cis(y), for finite nonzero y.

— `cexp(`$-\infty + i\infty$`)` returns $\pm 0 \pm i0$ (where the signs of the real and imaginary parts of the result are unspecified).

— `cexp(`$+\infty + i\infty$`)` returns $\pm\infty + i$NaN and raises the ‘‘invalid’’ floating-point exception (where the sign of the real part of the result is unspecified).

— `cexp(`$-\infty + i$NaN`)` returns $\pm 0 \pm i0$ (where the signs of the real and imaginary parts of the result are unspecified).

— `cexp(`$+\infty + i$NaN`)` returns $\pm\infty + i$NaN (where the sign of the real part of the result is unspecified).

— `cexp(`NaN $+ i0$`)` returns NaN $+ i0$.

— `cexp(`NaN $+ iy$`)` returns NaN $+ i$NaN and optionally raises the ‘‘invalid’’ floating-point exception, for all nonzero numbers y.

— `cexp(`NaN $+ i$NaN`)` returns NaN $+ i$NaN.

G.6.3.2 The `clog` functions

1 — `clog(conj(`z`))` = `conj(clog(`z`))`.

— `clog(`$-0 + i0$`)` returns $-\infty + i\pi$ and raises the ‘‘divide-by-zero’’ floating-point exception.

— `clog(`$+0 + i0$`)` returns $-\infty + i0$ and raises the ‘‘divide-by-zero’’ floating-point exception.

— `clog(`$x + i\infty$`)` returns $+\infty + i\pi/2$, for finite x.

— `clog(`$x + i$NaN`)` returns NaN $+ i$NaN and optionally raises the ‘‘invalid’’ floating-point exception, for finite x.

— **`clog(`**$-\infty + iy$**`)`** returns $+\infty + i\pi$, for finite positive-signed y.

— **`clog(`**$+\infty + iy$**`)`** returns $+\infty + i0$, for finite positive-signed y.

— **`clog(`**$-\infty + i\infty$**`)`** returns $+\infty + i3\pi/4$.

— **`clog(`**$+\infty + i\infty$**`)`** returns $+\infty + i\pi/4$.

— **`clog(`**$\pm\infty + i\mathrm{NaN}$**`)`** returns $+\infty + i\mathrm{NaN}$.

— **`clog(`**$\mathrm{NaN} + iy$**`)`** returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the ‘‘invalid’’ floating-point exception, for finite y.

— **`clog(`**$\mathrm{NaN} + i\infty$**`)`** returns $+\infty + i\mathrm{NaN}$.

— **`clog(`**$\mathrm{NaN} + i\mathrm{NaN}$**`)`** returns $\mathrm{NaN} + i\mathrm{NaN}$.

G.6.4 Power and absolute-value functions

G.6.4.1 The `cpow` functions

1 The **`cpow`** functions raise floating-point exceptions if appropriate for the calculation of the parts of the result, and may also raise spurious floating-point exceptions.[379)]

G.6.4.2 The `csqrt` functions

1 — **`csqrt(conj(`**z**`)) = conj(csqrt(`**z**`))`**.

— **`csqrt(`**$\pm 0 + i0$**`)`** returns $+0 + i0$.

— **`csqrt(`**$x + i\infty$**`)`** returns $+\infty + i\infty$, for all x (including NaN).

— **`csqrt(`**$x + i\mathrm{NaN}$**`)`** returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the ‘‘invalid’’ floating-point exception, for finite x.

— **`csqrt(`**$-\infty + iy$**`)`** returns $+0 + i\infty$, for finite positive-signed y.

— **`csqrt(`**$+\infty + iy$**`)`** returns $+\infty + i0$, for finite positive-signed y.

— **`csqrt(`**$-\infty + i\mathrm{NaN}$**`)`** returns $\mathrm{NaN} \pm i\infty$ (where the sign of the imaginary part of the result is unspecified).

— **`csqrt(`**$+\infty + i\mathrm{NaN}$**`)`** returns $+\infty + i\mathrm{NaN}$.

— **`csqrt(`**$\mathrm{NaN} + iy$**`)`** returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the ‘‘invalid’’ floating-point exception, for finite y.

— **`csqrt(`**$\mathrm{NaN} + i\mathrm{NaN}$**`)`** returns $\mathrm{NaN} + i\mathrm{NaN}$.

379) This allows **`cpow(`***`z`*`, `*`c`***`)`** to be implemented as **`cexp(`***`c`*` `**`clog(`***`z`***`))`** without precluding implementations that treat special cases more carefully.

G.7 Type-generic math `<tgmath.h>`

1 Type-generic macros that accept complex arguments also accept imaginary arguments. If an argument is imaginary, the macro expands to an expression whose type is real, imaginary, or complex, as appropriate for the particular function: if the argument is imaginary, then the types of **`cos`**, **`cosh`**, **`fabs`**, **`carg`**, **`cimag`**, and **`creal`** are real; the types of **`sin`**, **`tan`**, **`sinh`**, **`tanh`**, **`asin`**, **`atan`**, **`asinh`**, and **`atanh`** are imaginary; and the types of the others are complex.

2 Given an imaginary argument, each of the type-generic macros **`cos`**, **`sin`**, **`tan`**, **`cosh`**, **`sinh`**, **`tanh`**, **`asin`**, **`atan`**, **`asinh`**, **`atanh`** is specified by a formula in terms of real functions:

$$
\begin{aligned}
\mathtt{cos}(iy) &= \mathtt{cosh}(y)\\
\mathtt{sin}(iy) &= i\ \mathtt{sinh}(y)\\
\mathtt{tan}(iy) &= i\ \mathtt{tanh}(y)\\
\mathtt{cosh}(iy) &= \mathtt{cos}(y)\\
\mathtt{sinh}(iy) &= i\ \mathtt{sin}(y)\\
\mathtt{tanh}(iy) &= i\ \mathtt{tan}(y)\\
\mathtt{asin}(iy) &= i\ \mathtt{asinh}(y)\\
\mathtt{atan}(iy) &= i\ \mathtt{atanh}(y)\\
\mathtt{asinh}(iy) &= i\ \mathtt{asin}(y)\\
\mathtt{atanh}(iy) &= i\ \mathtt{atan}(y)
\end{aligned}
$$

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 **`<stdbool.h>`**.

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`**
gtrI	**`x > y`**
geqI	**`x >= y`**

where **`x`** and **`y`** are expressions of the same integer type.

H.2.3 Floating-point types

1 The C floating-point types **`float`**, **`double`**, and **`long double`** are compatible with LIA−1. If an implementation adds support for the LIA−1 exceptional values ‘‘underflow’’, ‘‘floating_overflow’’, and ‘‘"undefined’’, then those types are conformant with LIA−1. An implementation that uses IEC 60559 floating-point formats and operations (see annex F) along with IEC 60559 status flags and traps has LIA−1 conformant types.

H.2.3.1 Floating-point parameters

1 The parameters for a floating point data type can be accessed by the following:

r	**`FLT_RADIX`**
p	**`FLT_MANT_DIG`**, **`DBL_MANT_DIG`**, **`LDBL_MANT_DIG`**
emax	**`FLT_MAX_EXP`**, **`DBL_MAX_EXP`**, **`LDBL_MAX_EXP`**
emin	**`FLT_MIN_EXP`**, **`DBL_MIN_EXP`**, **`LDBL_MIN_EXP`**

2 The derived constants for the floating point types are accessed by the following:

fmax	**`FLT_MAX`**, **`DBL_MAX`**, **`LDBL_MAX`**
fminN	**`FLT_MIN`**, **`DBL_MIN`**, **`LDBL_MIN`**
epsilon	**`FLT_EPSILON`**, **`DBL_EPSILON`**, **`LDBL_EPSILON`**
rnd_style	**`FLT_ROUNDS`**

H.2.3.2 Floating-point operations

1 The floating-point operations on floating-point types are the following:

addF	**`x + y`**
subF	**`x - y`**
mulF	**`x * y`**
divF	**`x / y`**
negF	**`-x`**
absF	**`fabsf(x)`**, **`fabs(x)`**, **`fabsl(x)`**
exponentF	**`1.f+logbf(x)`**, **`1.0+logb(x)`**, **`1.L+logbl(x)`**
scaleF	**`scalbnf(x, n)`**, **`scalbn(x, n)`**, **`scalbnl(x, n)`**, **`scalblnf(x, li)`**, **`scalbln(x, li)`**, **`scalblnl(x, li)`**
intpartF	**`modff(x, &y)`**, **`modf(x, &y)`**, **`modfl(x, &y)`**
fractpartF	**`modff(x, &y)`**, **`modf(x, &y)`**, **`modfl(x, &y)`**
eqF	**`x == y`**
neqF	**`x != y`**
lssF	**`x < y`**
leqF	**`x <= y`**
gtrF	**`x > y`**
geqF	**`x >= y`**

where **`x`** and **`y`** are expressions of the same floating point type, **`n`** is of type **`int`**, and **`li`** is of type **`long int`**.

H.2.3.3 Rounding styles

1 The C Standard requires all floating types to use the same radix and rounding style, so that only one identifier for each is provided to map to LIA−1.

2 The **`FLT_ROUNDS`** parameter can be used to indicate the LIA−1 rounding styles:

truncate	**`FLT_ROUNDS == 0`**

nearest **`FLT_ROUNDS == 1`**

other **`FLT_ROUNDS != 0 && FLT_ROUNDS != 1`**

provided that an implementation extends **`FLT_ROUNDS`** to cover the rounding style used in all relevant LIA−1 operations, not just addition as in C.

H.2.4 Type conversions

1 The LIA−1 type conversions are the following type casts:

cvtI' → I **`(int)i`**, **`(long int)i`**, **`(long long int)i`**, **`(unsigned int)i`**, **`(unsigned long int)i`**, **`(unsigned long long int)i`**

cvtF → I **`(int)x`**, **`(long int)x`**, **`(long long int)x`**, **`(unsigned int)x`**, **`(unsigned long int)x`**, **`(unsigned long long int)x`**

cvtI → F **`(float)i`**, **`(double)i`**, **`(long double)i`**

cvtF' → F **`(float)x`**, **`(double)x`**, **`(long double)x`**

2 In the above conversions from floating to integer, the use of **(***cast***)x** can be replaced with **(***cast***)round(x)**, **(***cast***)rint(x)**, **(***cast***)nearbyint(x)**, **(***cast***)trunc(x)**, **(***cast***)ceil(x)**, or **(***cast***)floor(x)**. In addition, C's floating-point to integer conversion functions, **`lrint()`**, **`llrint()`**, **`lround()`**, and **`llround()`**, can be used. They all meet LIA−1's requirements on floating to integer rounding for in-range values. For out-of-range values, the conversions shall silently wrap for the modulo types.

3 The **`fmod()`** function is useful for doing silent wrapping to unsigned integer types, e.g., **`fmod( fabs(rint(x)), 65536.0 )`** or **`(0.0 <= (y = fmod( rint(x), 65536.0 )) ? y : 65536.0 + y)`** will compute an integer value in the range 0.0 to 65535.0 which can then be cast to **`unsigned short int`**. But, the **`remainder()`** function is not useful for doing silent wrapping to signed integer types, e.g., **`remainder( rint(x), 65536.0 )`** will compute an integer value in the range −32767.0 to +32768.0 which is not, in general, in the range of **`signed short int`**.

4 C's conversions (casts) from floating-point to floating-point can meet LIA−1 requirements if an implementation uses round-to-nearest (IEC 60559 default).

5 C's conversions (casts) from integer to floating-point can meet LIA−1 requirements if an implementation uses round-to-nearest.

H.3 Notification

1 Notification is the process by which a user or program is informed that an exceptional arithmetic operation has occurred. C's operations are compatible with LIA−1 in that C allows an implementation to cause a notification to occur when any arithmetic operation returns an exceptional value as defined in LIA−1 clause 5.

H.3.1 Notification alternatives

1 LIA−1 requires at least the following two alternatives for handling of notifications: setting indicators or trap-and-terminate. LIA−1 allows a third alternative: trap-and-resume.

2 An implementation need only support a given notification alternative for the entire program. An implementation may support the ability to switch between notification alternatives during execution, but is not required to do so. An implementation can provide separate selection for each kind of notification, but this is not required.

3 C allows an implementation to provide notification. C's **SIGFPE** (for traps) and **FE_INVALID**, **FE_DIVBYZERO**, **FE_OVERFLOW**, **FE_UNDERFLOW** (for indicators) can provide LIA−1 notification.

4 C's signal handlers are compatible with LIA−1. Default handling of **SIGFPE** can provide trap-and-terminate behavior, except for those LIA−1 operations implemented by math library function calls. User-provided signal handlers for **SIGFPE** allow for trap-and-resume behavior with the same constraint.

H.3.1.1 Indicators

1 C's **<fenv.h>** status flags are compatible with the LIA−1 indicators.

2 The following mapping is for floating-point types:

undefined	**FE_INVALID**, **FE_DIVBYZERO**
floating_overflow	**FE_OVERFLOW**
underflow	**FE_UNDERFLOW**

3 The floating-point indicator interrogation and manipulation operations are:

set_indicators	**feraiseexcept(i)**
clear_indicators	**feclearexcept(i)**
test_indicators	**fetestexcept(i)**
current_indicators	**fetestexcept(FE_ALL_EXCEPT)**

where **i** is an expression of type **int** representing a subset of the LIA−1 indicators.

4 C allows an implementation to provide the following LIA−1 required behavior: at program termination if any indicator is set the implementation shall send an unambiguous

and ‘‘hard to ignore’’ message (see LIA−1 subclause 6.1.2)

5 LIA−1 does not make the distinction between floating-point and integer for ‘‘undefined’’. This documentation makes that distinction because **`<fenv.h>`** covers only the floating-point indicators.

H.3.1.2 Traps

1 C is compatible with LIA−1’s trap requirements for arithmetic operations, but not for math library functions (which are not permitted to invoke a user’s signal handler for **`SIGFPE`**). An implementation can provide an alternative of notification through termination with a ‘‘hard-to-ignore’’ message (see LIA−1 subclause 6.1.3).

2 LIA−1 does not require that traps be precise.

3 C does require that **`SIGFPE`** be the signal corresponding to LIA−1 arithmetic exceptions, if there is any signal raised for them.

4 C supports signal handlers for **`SIGFPE`** and allows trapping of LIA−1 arithmetic exceptions. When LIA−1 arithmetic exceptions do trap, C’s signal-handler mechanism allows trap-and-terminate (either default implementation behavior or user replacement for it) or trap-and-resume, at the programmer’s option.

Annex I
(informative)
Common warnings

1 An implementation may generate warnings in many situations, none of which are specified as part of this International Standard. The following are a few of the more common situations.

2 — A new **`struct`** or **`union`** type appears in a function prototype (6.2.1, 6.7.2.3).

— A block with initialization of an object that has automatic storage duration is jumped into (6.2.4).

— An implicit narrowing conversion is encountered, such as the assignment of a **`long int`** or a **`double`** to an **`int`**, or a pointer to **`void`** to a pointer to any type other than a character type (6.3).

— A hexadecimal floating constant cannot be represented exactly in its evaluation format (6.4.4.2).

— An integer character constant includes more than one character or a wide character constant includes more than one multibyte character (6.4.4.4).

— The characters **`/*`** are found in a comment (6.4.7).

— An ‘‘unordered’’ binary operator (not comma, **`&&`**, or **`||`**) contains a side effect to an lvalue in one operand, and a side effect to, or an access to the value of, the identical lvalue in the other operand (6.5).

— A function is called but no prototype has been supplied (6.5.2.2).

— The arguments in a function call do not agree in number and type with those of the parameters in a function definition that is not a prototype (6.5.2.2).

— An object is defined but not used (6.7).

— A value is given to an object of an enumerated type other than by assignment of an enumeration constant that is a member of that type, or an enumeration object that has the same type, or the value of a function that returns the same enumerated type (6.7.2.2).

— An aggregate has a partly bracketed initialization (6.7.8).

— A statement cannot be reached (6.8).

— A statement with no apparent effect is encountered (6.8).

— A constant expression is used as the controlling expression of a selection statement (6.8.4).

— An incorrectly formed preprocessing group is encountered while skipping a preprocessing group (6.10.1).

— An unrecognized **`#pragma`** directive is encountered (6.10.6).

Annex J
(informative)
Portability issues

1 This annex collects some information about portability that appears in this International Standard.

J.1 Unspecified behavior

1 The following are unspecified:

— The manner and timing of static initialization (5.1.2).

— The termination status returned to the hosted environment if the return type of **`main`** is not compatible with int (5.1.2.2.3).

— The values of objects that are neither lock-free atomic objects nor of type **`volatile sig_atomic_t`** and the state of the floating-point environment, when the processing of the abstract machine is interrupted by receipt of a signal (5.1.2.3).

— The behavior of the display device if a printing character is written when the active position is at the final position of a line (5.2.2).

— The behavior of the display device if a backspace character is written when the active position is at the initial position of a line (5.2.2).

— The behavior of the display device if a horizontal tab character is written when the active position is at or past the last defined horizontal tabulation position (5.2.2).

— The behavior of the display device if a vertical tab character is written when the active position is at or past the last defined vertical tabulation position (5.2.2).

— How an extended source character that does not correspond to a universal character name counts toward the significant initial characters in an external identifier (5.2.4.1).

— Many aspects of the representations of types (6.2.6).

— The value of padding bytes when storing values in structures or unions (6.2.6.1).

— The values of bytes that correspond to union members other than the one last stored into (6.2.6.1).

— The representation used when storing a value in an object that has more than one object representation for that value (6.2.6.1).

— The values of any padding bits in integer representations (6.2.6.2).

— Whether certain operators can generate negative zeros and whether a negative zero becomes a normal zero when stored in an object (6.2.6.2).

— Whether two string literals result in distinct arrays (6.4.5).

— The order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call `()`, `&&`, `||`, `? :`, and comma operators (6.5).

— The order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call (6.5.2.2).

— The order of side effects among compound literal initialization list expressions (6.5.2.5).

— The order in which the operands of an assignment operator are evaluated (6.5.16).

— The alignment of the addressable storage unit allocated to hold a bit-field (6.7.2.1).

— Whether a call to an inline function uses the inline definition or the external definition of the function (6.7.4).

— Whether or not a size expression is evaluated when it is part of the operand of a **`sizeof`** operator and changing the value of the size expression would not affect the result of the operator (6.7.6.2).

— The order in which any side effects occur among the initialization list expressions in an initializer (6.7.9).

— The layout of storage for function parameters (6.9.1).

— When a fully expanded macro replacement list contains a function-like macro name as its last preprocessing token and the next preprocessing token from the source file is a `(`, and the fully expanded replacement of that macro ends with the name of the first macro and the next preprocessing token from the source file is again a `(`, whether that is considered a nested replacement (6.10.3).

— The order in which `#` and `##` operations are evaluated during macro substitution (6.10.3.2, 6.10.3.3).

— The state of the floating-point status flags when execution passes from a part of the program translated with **`FENV_ACCESS`** ‘‘off’’ to a part translated with **`FENV_ACCESS`** ‘‘on’’ (7.6.1).

— The order in which **`feraiseexcept`** raises floating-point exceptions, except as stated in F.8.6 (7.6.2.3).

— Whether **`math_errhandling`** is a macro or an identifier with external linkage (7.12).

— The results of the **`frexp`** functions when the specified value is not a floating-point number (7.12.6.4).

— The numeric result of the **ilogb** functions when the correct value is outside the range of the return type (7.12.6.5, F.10.3.5).

— The result of rounding when the value is out of range (7.12.9.5, 7.12.9.7, F.10.6.5).

— The value stored by the **remquo** functions in the object pointed to by **quo** when **y** is zero (7.12.10.3).

— Whether a comparison macro argument that is represented in a format wider than its semantic type is converted to the semantic type (7.12.14).

— Whether **setjmp** is a macro or an identifier with external linkage (7.13).

— Whether **va_copy** and **va_end** are macros or identifiers with external linkage (7.16.1).

— The hexadecimal digit before the decimal point when a non-normalized floating-point number is printed with an **a** or **A** conversion specifier (7.21.6.1, 7.29.2.1).

— The value of the file position indicator after a successful call to the **ungetc** function for a text stream, or the **ungetwc** function for any stream, until all pushed-back characters are read or discarded (7.21.7.10, 7.29.3.10).

— The details of the value stored by the **fgetpos** function (7.21.9.1).

— The details of the value returned by the **ftell** function for a text stream (7.21.9.4).

— Whether the **strtod**, **strtof**, **strtold**, **wcstod**, **wcstof**, and **wcstold** functions convert a minus-signed sequence to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (7.22.1.3, 7.29.4.1.1).

— The order and contiguity of storage allocated by successive calls to the **calloc**, **malloc**, and **realloc** functions (7.22.3).

— The amount of storage allocated by a successful call to the **calloc**, **malloc**, or **realloc** function when 0 bytes was requested (7.22.3).

— Whether a call to the **atexit** function that does not happen before the **exit** function is called will succeed (7.22.4.2).

— Whether a call to the **at_quick_exit** function that does not happen before the **quick_exit** function is called will succeed (7.22.4.3).

— Which of two elements that compare as equal is matched by the **bsearch** function (7.22.5.1).

— The order of two elements that compare as equal in an array sorted by the **qsort** function (7.22.5.2).

— The encoding of the calendar time returned by the **time** function (7.27.2.4).

— The characters stored by the **strftime** or **wcsftime** function if any of the time values being converted is outside the normal range (7.27.3.5, 7.29.5.1).

— Whether an encoding error occurs if a **wchar_t** value that does not correspond to a member of the extended character set appears in the format string for a function in 7.29.2 or 7.29.5 and the specified semantics do not require that value to be processed by **wcrtomb** (7.29.1).

— The conversion state after an encoding error occurs (7.29.6.3.2, 7.29.6.3.3, 7.29.6.4.1, 7.29.6.4.2,

— The resulting value when the ‘‘invalid’’ floating-point exception is raised during IEC 60559 floating to integer conversion (F.4).

— Whether conversion of non-integer IEC 60559 floating values to integer raises the ‘‘inexact’’ floating-point exception (F.4).

— Whether or when library functions in **<math.h>** raise the ‘‘inexact’’ floating-point exception in an IEC 60559 conformant implementation (F.10).

— Whether or when library functions in **<math.h>** raise an undeserved ‘‘underflow’’ floating-point exception in an IEC 60559 conformant implementation (F.10).

— The exponent value stored by **frexp** for a NaN or infinity (F.10.3.4).

— The numeric result returned by the **lrint**, **llrint**, **lround**, and **llround** functions if the rounded value is outside the range of the return type (F.10.6.5, F.10.6.7).

— The sign of one part of the **complex** result of several math functions for certain special cases in IEC 60559 compatible implementations (G.6.1.1, G.6.2.2, G.6.2.3, G.6.2.4, G.6.2.5, G.6.2.6, G.6.3.1, G.6.4.2).

J.2 Undefined behavior

1 The behavior is undefined in the following circumstances:

— A ‘‘shall’’ or ‘‘shall not’’ requirement that appears outside of a constraint is violated (clause 4).

— A nonempty source file does not end in a new-line character which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment (5.1.1.2).

— Token concatenation produces a character sequence matching the syntax of a universal character name (5.1.1.2).

— A program in a hosted environment does not define a function named **main** using one of the specified forms (5.1.2.2.1).

— The execution of a program contains a data race (5.1.2.4).

— A character not in the basic source character set is encountered in a source file, except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token (5.2.1).

— An identifier, comment, string literal, character constant, or header name contains an invalid multibyte character or does not begin and end in the initial shift state (5.2.1.2).

— The same identifier has both internal and external linkage in the same translation unit (6.2.2).

— An object is referred to outside of its lifetime (6.2.4).

— The value of a pointer to an object whose lifetime has ended is used (6.2.4).

— The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.9, 6.8).

— A trap representation is read by an lvalue expression that does not have character type (6.2.6.1).

— A trap representation is produced by a side effect that modifies any part of the object using an lvalue expression that does not have character type (6.2.6.1).

— The operands to certain operators are such that they could produce a negative zero result, but the implementation does not support negative zeros (6.2.6.2).

— Two declarations of the same object or function specify types that are not compatible (6.2.7).

— A program requires the formation of a composite type from a variable length array type whose size is specified by an expression that is not evaluated (6.2.7).

— Conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).

— Demotion of one real floating type to another produces a value outside the range that can be represented (6.3.1.5).

— An lvalue does not designate an object when evaluated (6.3.2.1).

— A non-array lvalue with an incomplete type is used in a context that requires the value of the designated object (6.3.2.1).

— An lvalue designating an object of automatic storage duration that could have been declared with the **`register`** storage class is used in a context that requires the value of the designated object, but the object is uninitialized. (6.3.2.1).

— An lvalue having array type is converted to a pointer to the initial element of the array, and the array object has register storage class (6.3.2.1).

— An attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to **void**) is applied to a void expression (6.3.2.2).

— Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).

— Conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3).

— A pointer is used to call a function whose type is not compatible with the referenced type (6.3.2.3).

— An unmatched **'** or **"** character is encountered on a logical source line during tokenization (6.4).

— A reserved keyword token is used in translation phase 7 or 8 for some purpose other than as a keyword (6.4.1).

— A universal character name in an identifier does not designate a character whose encoding falls into one of the specified ranges (6.4.2.1).

— The initial character of an identifier is a universal character name designating a digit (6.4.2.1).

— Two identifiers differ only in nonsignificant characters (6.4.2.1).

— The identifier **__func__** is explicitly declared (6.4.2.2).

— The program attempts to modify a string literal (6.4.5).

— The characters **'**, ****, **"**, **//**, or **/*** occur in the sequence between the **<** and **>** delimiters, or the characters **'**, ****, **//**, or **/*** occur in the sequence between the **"** delimiters, in a header name preprocessing token (6.4.7).

— A side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object (6.5).

— An exceptional condition occurs during the evaluation of an expression (6.5).

— An object has its stored value accessed other than by an lvalue of an allowable type (6.5).

— For a call to a function without a function prototype in scope, the number of arguments does not equal the number of parameters (6.5.2.2).

— For call to a function without a function prototype in scope where the function is defined with a function prototype, either the prototype ends with an ellipsis or the types of the arguments after promotion are not compatible with the types of the parameters (6.5.2.2).

— For a call to a function without a function prototype in scope where the function is not defined with a function prototype, the types of the arguments after promotion are not compatible with those of the parameters after promotion (with certain exceptions) (6.5.2.2).

— A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2).

— A member of an atomic structure or union is accessed (6.5.2.3).

— The operand of the unary **`*`** operator has an invalid value (6.5.3.2).

— A pointer is converted to other than an integer or pointer type (6.5.4).

— The value of the second operand of the **`/`** or **`%`** operator is zero (6.5.5).

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary **`*`** operator that is evaluated (6.5.6).

— Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).

— An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression **`a[1][7]`** given the declaration **`int a[4][5]`**) (6.5.6).

— The result of subtracting two pointers is not representable in an object of type **`ptrdiff_t`** (6.5.6).

— An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7).

— An expression having signed promoted type is left-shifted and either the value of the expression is negative or the result of shifting would be not be representable in the promoted type (6.5.7).

— Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators (6.5.8).

— An object is assigned to an inexactly overlapping object or to an exactly overlapping object with incompatible type (6.5.16.1).

— An expression that is required to be an integer constant expression does not have an integer type; has operands that are not integer constants, enumeration constants, character constants, **`sizeof`** expressions whose results are integer constants,

_Alignof expressions, or immediately-cast floating constants; or contains casts (outside operands to **sizeof** and **_Alignof** operators) other than conversions of arithmetic types to integer types (6.6).

— A constant expression in an initializer is not, or does not evaluate to, one of the following: an arithmetic constant expression, a null pointer constant, an address constant, or an address constant for a complete object type plus or minus an integer constant expression (6.6).

— An arithmetic constant expression does not have arithmetic type; has operands that are not integer constants, floating constants, enumeration constants, character constants, **sizeof** expressions whose results are integer constants, or **_Alignof** expressions; or contains casts (outside operands to **sizeof** or **_Alignof** operators) other than conversions of arithmetic types to arithmetic types (6.6).

— The value of an object is accessed by an array-subscript **[]**, member-access **.** or **->**, address **&**, or indirection ***** operator or a pointer cast in creating an address constant (6.6).

— An identifier for an object is declared with no linkage and the type of the object is incomplete after its declarator, or after its init-declarator if it has an initializer (6.7).

— A function is declared at block scope with an explicit storage-class specifier other than **extern** (6.7.1).

— A structure or union is defined without any named members (including those specified indirectly via anonymous structures and unions) (6.7.2.1).

— An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array (6.7.2.1).

— When the complete type is needed, an incomplete structure or union type is not completed in the same scope by another declaration of the tag that defines the content (6.7.2.3).

— An attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type (6.7.3).

— An attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type (6.7.3).

— The specification of a function type includes any type qualifiers (6.7.3).

— Two qualified types that are required to be compatible do not have the identically qualified version of a compatible type (6.7.3).

— An object which has been modified is accessed through a restrict-qualified pointer to a const-qualified type, or through a restrict-qualified pointer and another pointer that are not both based on the same object (6.7.3.1).

— A restrict-qualified pointer is assigned a value based on another restricted pointer whose associated block neither began execution before the block associated with this pointer, nor ended before the assignment (6.7.3.1).

— A function with external linkage is declared with an **`inline`** function specifier, but is not also defined in the same translation unit (6.7.4).

— A function declared with a **`_Noreturn`** function specifier returns to its caller (6.7.4).

— The definition of an object has an alignment specifier and another declaration of that object has a different alignment specifier (6.7.5).

— Declarations of an object in different translation units have different alignment specifiers (6.7.5).

— Two pointer types that are required to be compatible are not identically qualified, or are not pointers to compatible types (6.7.6.1).

— The size expression in an array declaration is not a constant expression and evaluates at program execution time to a nonpositive value (6.7.6.2).

— In a context requiring two array types to be compatible, they do not have compatible element types, or their size specifiers evaluate to unequal values (6.7.6.2).

— A declaration of an array parameter includes the keyword **`static`** within the **`[`** and **`]`** and the corresponding argument does not provide access to the first element of an array with at least the specified number of elements (6.7.6.3).

— A storage-class specifier or type qualifier modifies the keyword **`void`** as a function parameter type list (6.7.6.3).

— In a context requiring two function types to be compatible, they do not have compatible return types, or their parameters disagree in use of the ellipsis terminator or the number and type of parameters (after default argument promotion, when there is no parameter type list or when one type is specified by a function definition with an identifier list) (6.7.6.3).

— The value of an unnamed member of a structure or union is used (6.7.9).

— The initializer for a scalar is neither a single expression nor a single expression enclosed in braces (6.7.9).

— The initializer for a structure or union object that has automatic storage duration is neither an initializer list nor a single expression that has compatible structure or union type (6.7.9).

— The initializer for an aggregate or union, other than an array initialized by a string literal, is not a brace-enclosed list of initializers for its elements or members (6.7.9).

— An identifier with external linkage is used, but in the program there does not exist exactly one external definition for the identifier, or the identifier is not used and there exist multiple external definitions for the identifier (6.9).

— A function definition includes an identifier list, but the types of the parameters are not declared in a following declaration list (6.9.1).

— An adjusted parameter type in a function definition is not a complete object type (6.9.1).

— A function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation (6.9.1).

— The **`}`** that terminates a function is reached, and the value of the function call is used by the caller (6.9.1).

— An identifier for an object with internal linkage and an incomplete type is declared with a tentative definition (6.9.2).

— The token **`defined`** is generated during the expansion of a **`#if`** or **`#elif`** preprocessing directive, or the use of the **`defined`** unary operator does not match one of the two specified forms prior to macro replacement (6.10.1).

— The **`#include`** preprocessing directive that results after expansion does not match one of the two header name forms (6.10.2).

— The character sequence in an **`#include`** preprocessing directive does not start with a letter (6.10.2).

— There are sequences of preprocessing tokens within the list of macro arguments that would otherwise act as preprocessing directives (6.10.3).

— The result of the preprocessing operator **`#`** is not a valid character string literal (6.10.3.2).

— The result of the preprocessing operator **`##`** is not a valid preprocessing token (6.10.3.3).

— The **`#line`** preprocessing directive that results after expansion does not match one of the two well-defined forms, or its digit sequence specifies zero or a number greater than 2147483647 (6.10.4).

— A non-**`STDC #pragma`** preprocessing directive that is documented as causing translation failure or some other form of undefined behavior is encountered (6.10.6).

— A **`#pragma STDC`** preprocessing directive does not match one of the well-defined forms (6.10.6).

— The name of a predefined macro, or the identifier **`defined`**, is the subject of a **`#define`** or **`#undef`** preprocessing directive (6.10.8).

— An attempt is made to copy an object to an overlapping object by use of a library function, other than as explicitly allowed (e.g., **memmove**) (clause 7).

— A file with the same name as one of the standard headers, not provided as part of the implementation, is placed in any of the standard places that are searched for included source files (7.1.2).

— A header is included within an external declaration or definition (7.1.2).

— A function, object, type, or macro that is specified as being declared or defined by some standard header is used before any header that declares or defines it is included (7.1.2).

— A standard header is included while a macro is defined with the same name as a keyword (7.1.2).

— The program attempts to declare a library function itself, rather than via a standard header, but the declaration does not have external linkage (7.1.2).

— The program declares or defines a reserved identifier, other than as allowed by 7.1.4 (7.1.3).

— The program removes the definition of a macro whose name begins with an underscore and either an uppercase letter or another underscore (7.1.3).

— An argument to a library function has an invalid value or a type not expected by a function with variable number of arguments (7.1.4).

— The pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid (7.1.4).

— The macro definition of **assert** is suppressed in order to access an actual function (7.2).

— The argument to the **assert** macro does not have a scalar type (7.2).

— The **CX_LIMITED_RANGE**, **FENV_ACCESS**, or **FP_CONTRACT** pragma is used in any context other than outside all external declarations or preceding all explicit declarations and statements inside a compound statement (7.3.4, 7.6.1, 7.12.2).

— The value of an argument to a character handling function is neither equal to the value of **EOF** nor representable as an **unsigned char** (7.4).

— A macro definition of **errno** is suppressed in order to access an actual object, or the program defines an identifier with the name **errno** (7.5).

— Part of the program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the **FENV_ACCESS** pragma ‘‘off’’ (7.6.1).

— The exception-mask argument for one of the functions that provide access to the floating-point status flags has a nonzero value not obtained by bitwise OR of the floating-point exception macros (7.6.2).

— The **fesetexceptflag** function is used to set floating-point status flags that were not specified in the call to the **fegetexceptflag** function that provided the value of the corresponding **fexcept_t** object (7.6.2.4).

— The argument to **fesetenv** or **feupdateenv** is neither an object set by a call to **fegetenv** or **feholdexcept**, nor is it an environment macro (7.6.4.3, 7.6.4.4).

— The value of the result of an integer arithmetic or conversion function cannot be represented (7.8.2.1, 7.8.2.2, 7.8.2.3, 7.8.2.4, 7.22.6.1, 7.22.6.2, 7.22.1).

— The program modifies the string pointed to by the value returned by the **setlocale** function (7.11.1.1).

— The program modifies the structure pointed to by the value returned by the **localeconv** function (7.11.2.1).

— A macro definition of **math_errhandling** is suppressed or the program defines an identifier with the name **math_errhandling** (7.12).

— An argument to a floating-point classification or comparison macro is not of real floating type (7.12.3, 7.12.14).

— A macro definition of **setjmp** is suppressed in order to access an actual function, or the program defines an external identifier with the name **setjmp** (7.13).

— An invocation of the **setjmp** macro occurs other than in an allowed context (7.13.2.1).

— The **longjmp** function is invoked to restore a nonexistent environment (7.13.2.1).

— After a **longjmp**, there is an attempt to access the value of an object of automatic storage duration that does not have volatile-qualified type, local to the function containing the invocation of the corresponding **setjmp** macro, that was changed between the **setjmp** invocation and **longjmp** call (7.13.2.1).

— The program specifies an invalid pointer to a signal handler function (7.14.1.1).

— A signal handler returns when the signal corresponded to a computational exception (7.14.1.1).

— A signal handler called in response to **SIGFPE**, **SIGILL**, **SIGSEGV**, or any other implementation-defined value corresponding to a computational exception returns (7.14.1.1).

— A signal occurs as the result of calling the **abort** or **raise** function, and the signal handler calls the **raise** function (7.14.1.1).

— A signal occurs other than as the result of calling the **abort** or **raise** function, and the signal handler refers to an 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 calls any function in the standard library other than the **abort** function, the **_Exit** function, the **quick_exit** function, or the **signal** function (for the same signal number) (7.14.1.1).

— The value of **errno** is referred to after a signal occurred other than as the result of calling the **abort** or **raise** function and the corresponding signal handler obtained a **SIG_ERR** return from a call to the **signal** function (7.14.1.1).

— A signal is generated by an asynchronous signal handler (7.14.1.1).

— The **signal** function is used in a multi-threaded program (7.14.1.1).

— A function with a variable number of arguments attempts to access its varying arguments other than through a properly declared and initialized **va_list** object, or before the **va_start** macro is invoked (7.16, 7.16.1.1, 7.16.1.4).

— The macro **va_arg** is invoked using the parameter **ap** that was passed to a function that invoked the macro **va_arg** with the same parameter (7.16).

— A macro definition of **va_start**, **va_arg**, **va_copy**, or **va_end** is suppressed in order to access an actual function, or the program defines an external identifier with the name **va_copy** or **va_end** (7.16.1).

— The **va_start** or **va_copy** macro is invoked without a corresponding invocation of the **va_end** macro in the same function, or vice versa (7.16.1, 7.16.1.2, 7.16.1.3, 7.16.1.4).

— The type parameter to the **va_arg** macro is not such that a pointer to an object of that type can be obtained simply by postfixing a ***** (7.16.1.1).

— The **va_arg** macro is invoked when there is no actual next argument, or with a specified type that is not compatible with the promoted type of the actual next argument, with certain exceptions (7.16.1.1).

— The **va_copy** or **va_start** macro is called to initialize a **va_list** that was previously initialized by either macro without an intervening invocation of the **va_end** macro for the same **va_list** (7.16.1.2, 7.16.1.4).

— The parameter *parmN* of a **va_start** macro is declared with the **register** storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions (7.16.1.4).

— The member designator parameter of an **offsetof** macro is an invalid right operand of the **.** operator for the *type* parameter, or designates a bit-field (7.19).

— The argument in an instance of one of the integer-constant macros is not a decimal, octal, or hexadecimal constant, or it has a value that exceeds the limits for the corresponding type (7.20.4).

— A byte input/output function is applied to a wide-oriented stream, or a wide character input/output function is applied to a byte-oriented stream (7.21.2).

— Use is made of any portion of a file beyond the most recent wide character written to a wide-oriented stream (7.21.2).

— The value of a pointer to a **`FILE`** object is used after the associated file is closed (7.21.3).

— The stream for the **`fflush`** function points to an input stream or to an update stream in which the most recent operation was input (7.21.5.2).

— The string pointed to by the **`mode`** argument in a call to the **`fopen`** function does not exactly match one of the specified character sequences (7.21.5.3).

— An output operation on an update stream is followed by an input operation without an intervening call to the **`fflush`** function or a file positioning function, or an input operation on an update stream is followed by an output operation with an intervening call to a file positioning function (7.21.5.3).

— An attempt is made to use the contents of the array that was supplied in a call to the **`setvbuf`** function (7.21.5.6).

— There are insufficient arguments for the format in a call to one of the formatted input/output functions, or an argument does not have an appropriate type (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

— The format in a call to one of the formatted input/output functions or to the **`strftime`** or **`wcsftime`** function is not a valid multibyte character sequence that begins and ends in its initial shift state (7.21.6.1, 7.21.6.2, 7.27.3.5, 7.29.2.1, 7.29.2.2, 7.29.5.1).

— In a call to one of the formatted output functions, a precision appears with a conversion specifier other than those described (7.21.6.1, 7.29.2.1).

— A conversion specification for a formatted output function uses an asterisk to denote an argument-supplied field width or precision, but the corresponding argument is not provided (7.21.6.1, 7.29.2.1).

— A conversion specification for a formatted output function uses a **`#`** or **`0`** flag with a conversion specifier other than those described (7.21.6.1, 7.29.2.1).

— A conversion specification for one of the formatted input/output functions uses a length modifier with a conversion specifier other than those described (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

— An **s** conversion specifier is encountered by one of the formatted output functions, and the argument is missing the null terminator (unless a precision is specified that does not require null termination) (7.21.6.1, 7.29.2.1).

— An **n** conversion specification for one of the formatted input/output functions includes any flags, an assignment-suppressing character, a field width, or a precision (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

— A **%** conversion specifier is encountered by one of the formatted input/output functions, but the complete conversion specification is not exactly **%%** (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

— An invalid conversion specification is found in the format for one of the formatted input/output functions, or the **strftime** or **wcsftime** function (7.21.6.1, 7.21.6.2, 7.27.3.5, 7.29.2.1, 7.29.2.2, 7.29.5.1).

— The number of characters or wide characters transmitted by a formatted output function (or written to an array, or that would have been written to an array) is greater than **INT_MAX** (7.21.6.1, 7.29.2.1).

— The number of input items assigned by a formatted input function is greater than **INT_MAX** (7.21.6.2, 7.29.2.2).

— The result of a conversion by one of the formatted input functions cannot be represented in the corresponding object, or the receiving object does not have an appropriate type (7.21.6.2, 7.29.2.2).

— A **c**, **s**, or **[** conversion specifier is encountered by one of the formatted input functions, and the array pointed to by the corresponding argument is not large enough to accept the input sequence (and a null terminator if the conversion specifier is **s** or **[**) (7.21.6.2, 7.29.2.2).

— A **c**, **s**, or **[** conversion specifier with an **l** qualifier is encountered by one of the formatted input functions, but the input is not a valid multibyte character sequence that begins in the initial shift state (7.21.6.2, 7.29.2.2).

— The input item for a **%p** conversion by one of the formatted input functions is not a value converted earlier during the same program execution (7.21.6.2, 7.29.2.2).

— The **vfprintf**, **vfscanf**, **vprintf**, **vscanf**, **vsnprintf**, **vsprintf**, **vsscanf**, **vfwprintf**, **vfwscanf**, **vswprintf**, **vswscanf**, **vwprintf**, or **vwscanf** function is called with an improperly initialized **va_list** argument, or the argument is used (other than in an invocation of **va_end**) after the function returns (7.21.6.8, 7.21.6.9, 7.21.6.10, 7.21.6.11, 7.21.6.12, 7.21.6.13, 7.21.6.14, 7.29.2.5, 7.29.2.6, 7.29.2.7, 7.29.2.8, 7.29.2.9, 7.29.2.10).

— The contents of the array supplied in a call to the **fgets** or **fgetws** function are used after a read error occurred (7.21.7.2, 7.29.3.2).

— The file position indicator for a binary stream is used after a call to the **ungetc** function where its value was zero before the call (7.21.7.10).

— The file position indicator for a stream is used after an error occurred during a call to the **fread** or **fwrite** function (7.21.8.1, 7.21.8.2).

— A partial element read by a call to the **fread** function is used (7.21.8.1).

— The **fseek** function is called for a text stream with a nonzero offset and either the offset was not returned by a previous successful call to the **ftell** function on a stream associated with the same file or **whence** is not **SEEK_SET** (7.21.9.2).

— The **fsetpos** function is called to set a position that was not returned by a previous successful call to the **fgetpos** function on a stream associated with the same file (7.21.9.3).

— A non-null pointer returned by a call to the **calloc**, **malloc**, or **realloc** function with a zero requested size is used to access an object (7.22.3).

— The value of a pointer that refers to space deallocated by a call to the **free** or **realloc** function is used (7.22.3).

— The alignment requested of the **aligned_alloc** function is not valid or not supported by the implementation, or the size requested is not an integral multiple of the alignment (7.22.3.1).

— The pointer argument to the **free** or **realloc** function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to **free** or **realloc** (7.22.3.3, 7.22.3.5).

— The value of the object allocated by the **malloc** function is used (7.22.3.4).

— The value of any bytes in a new object allocated by the **realloc** function beyond the size of the old object are used (7.22.3.5).

— The program calls the **exit** or **quick_exit** function more than once, or calls both functions (7.22.4.4, 7.22.4.7).

— During the call to a function registered with the **atexit** or **at_quick_exit** function, a call is made to the **longjmp** function that would terminate the call to the registered function (7.22.4.4, 7.22.4.7).

— The string set up by the **getenv** or **strerror** function is modified by the program (7.22.4.6, 7.24.6.2).

— A signal is raised while the **quick_exit** function is executing (7.22.4.7).

— A command is executed through the **system** function in a way that is documented as causing termination or some other form of undefined behavior (7.22.4.8).

— A searching or sorting utility function is called with an invalid pointer argument, even if the number of elements is zero (7.22.5).

— The comparison function called by a searching or sorting utility function alters the contents of the array being searched or sorted, or returns ordering values inconsistently (7.22.5).

— The array being searched by the **bsearch** function does not have its elements in proper order (7.22.5.1).

— The current conversion state is used by a multibyte/wide character conversion function after changing the **LC_CTYPE** category (7.22.7).

— A string or wide string utility function is instructed to access an array beyond the end of an object (7.24.1, 7.29.4).

— A string or wide string utility function is called with an invalid pointer argument, even if the length is zero (7.24.1, 7.29.4).

— The contents of the destination array are used after a call to the **strxfrm**, **strftime**, **wcsxfrm**, or **wcsftime** function in which the specified length was too small to hold the entire null-terminated result (7.24.4.5, 7.27.3.5, 7.29.4.4.4, 7.29.5.1).

— The first argument in the very first call to the **strtok** or **wcstok** is a null pointer (7.24.5.8, 7.29.4.5.7).

— The type of an argument to a type-generic macro is not compatible with the type of the corresponding parameter of the selected function (7.25).

— A complex argument is supplied for a generic parameter of a type-generic macro that has no corresponding complex function (7.25).

— At least one member of the broken-down time passed to **asctime** contains a value outside its normal range, or the calculated year exceeds four digits or is less than the year 1000 (7.27.3.1).

— The argument corresponding to an **s** specifier without an **l** qualifier in a call to the **fwprintf** function does not point to a valid multibyte character sequence that begins in the initial shift state (7.29.2.11).

— In a call to the **wcstok** function, the object pointed to by **ptr** does not have the value stored by the previous call for the same wide string (7.29.4.5.7).

— An **mbstate_t** object is used inappropriately (7.29.6).

— The value of an argument of type **wint_t** to a wide character classification or case mapping function is neither equal to the value of **WEOF** nor representable as a **wchar_t** (7.30.1).

— The **`iswctype`** function is called using a different **`LC_CTYPE`** category from the one in effect for the call to the **`wctype`** function that returned the description (7.30.2.2.1).

— The **`towctrans`** function is called using a different **`LC_CTYPE`** category from the one in effect for the call to the **`wctrans`** function that returned the description (7.30.3.2.1).

J.3 Implementation-defined behavior

1 A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:

J.3.1 Translation

1 — How a diagnostic is identified (3.10, 5.1.1.3).

— Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

J.3.2 Environment

1 — The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2).

— The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

— The effect of program termination in a freestanding environment (5.1.2.1).

— An alternative manner in which the **`main`** function may be defined (5.1.2.2.1).

— The values given to the strings pointed to by the **`argv`** argument to **`main`** (5.1.2.2.1).

— What constitutes an interactive device (5.1.2.3).

— Whether a program can have more than one thread of execution in a freestanding environment (5.1.2.4).

— The set of signals, their semantics, and their default handling (7.14).

— Signal values other than **`SIGFPE`**, **`SIGILL`**, and **`SIGSEGV`** that correspond to a computational exception (7.14.1.1).

— Signals for which the equivalent of **`signal(sig, SIG_IGN);`** is executed at program startup (7.14.1.1).

— The set of environment names and the method for altering the environment list used by the **`getenv`** function (7.22.4.6).

— The manner of execution of the string by the **`system`** function (7.22.4.8).

J.3.3 Identifiers

1 — Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

— The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

J.3.4 Characters

1 — The number of bits in a byte (3.6).

— The values of the members of the execution character set (5.2.1).

— The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

— The value of a **char** object into which has been stored any character other than a member of the basic execution character set (6.2.5).

— Which of **signed char** or **unsigned char** has the same range, representation, and behavior as ‘‘plain’’ **char** (6.2.5, 6.3.1.1).

— The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

— The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

— The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

— The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

— Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence (6.4.5).

— The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).

— The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).

— The encoding of any of **wchar_t**, **char16_t**, and **char32_t** where the corresponding standard encoding macro (**__STDC_ISO_10646__**, **__STDC_UTF_16__**, or **__STDC_UTF_32__**) is not defined (6.10.8.2).

J.3.5 Integers

1 — Any extended integer types that exist in the implementation (6.2.5).

— Whether signed integer types are represented using sign and magnitude, two's complement, or ones' complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).

— The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

— The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

— The results of some bitwise operations on signed integers (6.5).

J.3.6 Floating point

1 — The accuracy of the floating-point operations and of the library functions in **<math.h>** and **<complex.h>** that return floating-point results (5.2.4.2.2).

— The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in **<stdio.h>**, **<stdlib.h>**, and **<wchar.h>** (5.2.4.2.2).

— The rounding behaviors characterized by non-standard values of **FLT_ROUNDS** (5.2.4.2.2).

— The evaluation methods characterized by non-standard negative values of **FLT_EVAL_METHOD** (5.2.4.2.2).

— The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

— The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).

— How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

— Whether and how floating expressions are contracted when not disallowed by the **FP_CONTRACT** pragma (6.5).

— The default state for the **FENV_ACCESS** pragma (7.6.1).

— Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).

— The default state for the **FP_CONTRACT** pragma (7.12.2).

J.3.7 Arrays and pointers

1 — The result of converting a pointer to an integer or vice versa (6.3.2.3).

— The size of the result of subtracting two pointers to elements of the same array (6.5.6).

J.3.8 Hints

1 — The extent to which suggestions made by using the **`register`** storage-class specifier are effective (6.7.1).

— The extent to which suggestions made by using the **`inline`** function specifier are effective (6.7.4).

J.3.9 Structures, unions, enumerations, and bit-fields

1 — Whether a ''plain'' **`int`** bit-field is treated as a **`signed int`** bit-field or as an **`unsigned int`** bit-field (6.7.2, 6.7.2.1).

— Allowable bit-field types other than **`_Bool`**, **`signed int`**, and **`unsigned int`** (6.7.2.1).

— Whether atomic types are permitted for bit-fields (6.7.2.1).

— Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).

— The order of allocation of bit-fields within a unit (6.7.2.1).

— The alignment of non-bit-field members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

— The integer type compatible with each enumerated type (6.7.2.2).

J.3.10 Qualifiers

1 — What constitutes an access to an object that has volatile-qualified type (6.7.3).

J.3.11 Preprocessing directives

1 — The locations within **`#pragma`** directives where header name preprocessing tokens are recognized (6.4, 6.4.7).

— How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

— Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

— Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

— The places that are searched for an included `< >` delimited header, and how the places are specified or the header is identified (6.10.2).

— How the named source file is searched for in an included `" "` delimited header (6.10.2).

— The method by which preprocessing tokens (possibly resulting from macro expansion) in a `#include` directive are combined into a header name (6.10.2).

— The nesting limit for `#include` processing (6.10.2).

— Whether the `#` operator inserts a `\` character before the `\` character that begins a universal character name in a character constant or string literal (6.10.3.2).

— The behavior on each recognized non-`STDC #pragma` directive (6.10.6).

— The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (6.10.8.1).

J.3.12 Library functions

1 — Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).

— The format of the diagnostic printed by the `assert` macro (7.2.1.1).

— The representation of the floating-point status flags stored by the `fegetexceptflag` function (7.6.2.2).

— Whether the `feraiseexcept` function raises the ‘‘inexact’’ floating-point exception in addition to the ‘‘overflow’’ or ‘‘underflow’’ floating-point exception (7.6.2.3).

— Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function (7.11.1.1).

— The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 (7.12).

— Domain errors for the mathematics functions, other than those required by this International Standard (7.12.1).

— The values returned by the mathematics functions on domain errors or pole errors (7.12.1).

— The values returned by the mathematics functions on underflow range errors, whether `errno` is set to the value of the macro `ERANGE` when the integer expression `math_errhandling & MATH_ERRNO` is nonzero, and whether the ‘‘underflow’’ floating-point exception is raised when the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero. (7.12.1).

— Whether a domain error occurs or zero is returned when an **`fmod`** function has a second argument of zero (7.12.10.1).

— Whether a domain error occurs or zero is returned when a **`remainder`** function has a second argument of zero (7.12.10.2).

— The base-2 logarithm of the modulus used by the **`remquo`** functions in reducing the quotient (7.12.10.3).

— Whether a domain error occurs or zero is returned when a **`remquo`** function has a second argument of zero (7.12.10.3).

— Whether the equivalent of **`signal(sig, SIG_DFL);`** is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

— The null pointer constant to which the macro **`NULL`** expands (7.19).

— Whether the last line of a text stream requires a terminating new-line character (7.21.2).

— Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.21.2).

— The number of null characters that may be appended to data written to a binary stream (7.21.2).

— Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.21.3).

— Whether a write on a text stream causes the associated file to be truncated beyond that point (7.21.3).

— The characteristics of file buffering (7.21.3).

— Whether a zero-length file actually exists (7.21.3).

— The rules for composing valid file names (7.21.3).

— Whether the same file can be simultaneously open multiple times (7.21.3).

— The nature and choice of encodings used for multibyte characters in files (7.21.3).

— The effect of the **`remove`** function on an open file (7.21.4.1).

— The effect if a file with the new name exists prior to a call to the **`rename`** function (7.21.4.2).

— Whether an open temporary file is removed upon abnormal program termination (7.21.4.3).

— Which changes of mode are permitted (if any), and under what circumstances (7.21.5.4).

— The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.21.6.1, 7.29.2.1).

— The output for **%p** conversion in the **fprintf** or **fwprintf** function (7.21.6.1, 7.29.2.1).

— The interpretation of a **-** character that is neither the first nor the last character, nor the second where a **^** character is the first, in the scanlist for **%[** conversion in the **fscanf** or **fwscanf** function (7.21.6.2, 7.29.2.1).

— The set of sequences matched by a **%p** conversion and the interpretation of the corresponding input item in the **fscanf** or **fwscanf** function (7.21.6.2, 7.29.2.2).

— The value to which the macro **errno** is set by the **fgetpos**, **fsetpos**, or **ftell** functions on failure (7.21.9.1, 7.21.9.3, 7.21.9.4).

— The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the **strtod**, **strtof**, **strtold**, **wcstod**, **wcstof**, or **wcstold** function (7.22.1.3, 7.29.4.1.1).

— Whether or not the **strtod**, **strtof**, **strtold**, **wcstod**, **wcstof**, or **wcstold** function sets **errno** to **ERANGE** when underflow occurs (7.22.1.3, 7.29.4.1.1).

— Whether the **calloc**, **malloc**, and **realloc** functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.22.3).

— Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the **abort** or **_Exit** function is called (7.22.4.1, 7.22.4.5).

— The termination status returned to the host environment by the **abort**, **exit**, **_Exit**, or **quick_exit** function (7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7).

— The value returned by the **system** function when its argument is not a null pointer (7.22.4.8).

— The range and precision of times representable in **clock_t** and **time_t** (7.27). *

— The local time zone and Daylight Saving Time (7.27.1).

— The era for the **clock** function (7.27.2.1).

— The **TIME_UTC** epoch (7.27.2.5).

— The replacement string for the **%Z** specifier to the **strftime**, and **wcsftime** functions in the **"C"** locale (7.27.3.5, 7.29.5.1).

— Whether the functions in **<math.h>** honor the rounding direction mode in an IEC 60559 conformant implementation, unless explicitly specified otherwise (F.10).

J.3.13 Architecture

1 — The values or expressions assigned to the macros specified in the headers **<float.h>**, **<limits.h>**, and **<stdint.h>** (5.2.4.2, 7.20.2, 7.20.3).

— The result of attempting to indirectly access an object with automatic or thread storage duration from a thread other than the one with which it is associated (6.2.4).

— The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).

— Whether any extended alignments are supported and the contexts in which they are supported (6.2.8).

— Valid alignment values other than those returned by an **_Alignof** expression for fundamental types, if any (6.2.8).

— The value of the result of the **sizeof** and **_Alignof** operators (6.5.3.4).

J.4 Locale-specific behavior

1 The following characteristics of a hosted environment are locale-specific and are required to be documented by the implementation:

— Additional members of the source and execution character sets beyond the basic character set (5.2.1).

— The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.2).

— The shift states used for the encoding of multibyte characters (5.2.1.2).

— The direction of writing of successive printing characters (5.2.2).

— The decimal-point character (7.1.1).

— The set of printing characters (7.4, 7.30.2).

— The set of control characters (7.4, 7.30.2).

— The sets of characters tested for by the **isalpha**, **isblank**, **islower**, **ispunct**, **isspace**, **isupper**, **iswalpha**, **iswblank**, **iswlower**, **iswpunct**, **iswspace**, or **iswupper** functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.30.2.1.2, 7.30.2.1.3, 7.30.2.1.7, 7.30.2.1.9, 7.30.2.1.10, 7.30.2.1.11).

— The native environment (7.11.1.1).

— Additional subject sequences accepted by the numeric conversion functions (7.22.1, 7.29.4.1).

— The collation sequence of the execution character set (7.24.4.3, 7.29.4.4.2).

— The contents of the error message strings set up by the **`strerror`** function (7.24.6.2).

— The formats for time and date (7.27.3.5, 7.29.5.1).

— Character mappings that are supported by the **`towctrans`** function (7.30.1).

— Character classifications that are supported by the **`iswctype`** function (7.30.1).

J.5 Common extensions

1 The following extensions are widely used in many systems, but are not portable to all implementations. The inclusion of any extension that may cause a strictly conforming program to become invalid renders an implementation nonconforming. Examples of such extensions are new keywords, extra library functions declared in standard headers, or predefined macros with names that do not begin with an underscore.

J.5.1 Environment arguments

1 In a hosted environment, the **`main`** function receives a third argument, **`char *envp[]`**, that points to a null-terminated array of pointers to **`char`**, each of which points to a string that provides information about the environment for this execution of the program (5.1.2.2.1).

J.5.2 Specialized identifiers

1 Characters other than the underscore _, letters, and digits, that are not part of the basic source character set (such as the dollar sign **`$`**, or characters in national character sets) may appear in an identifier (6.4.2).

J.5.3 Lengths and cases of identifiers

1 All characters in identifiers (with or without external linkage) are significant (6.4.2).

J.5.4 Scopes of identifiers

1 A function identifier, or the identifier of an object the declaration of which contains the keyword **`extern`**, has file scope (6.2.1).

J.5.5 Writable string literals

1 String literals are modifiable (in which case, identical string literals should denote distinct objects) (6.4.5).

J.5.6 Other arithmetic types

1 Additional arithmetic types, such as **`__int128`** or **`double double`**, and their appropriate conversions are defined (6.2.5, 6.3.1). Additional floating types may have more range or precision than **`long double`**, may be used for evaluating expressions of other floating types, and may be used to define **`float_t`** or **`double_t`**. Additional floating types may also have less range or precision than **`float`**.

J.5.7 Function pointer casts

1 A pointer to an object or to **`void`** may be cast to a pointer to a function, allowing data to be invoked as a function (6.5.4).

2 A pointer to a function may be cast to a pointer to an object or to **`void`**, allowing a function to be inspected or modified (for example, by a debugger) (6.5.4).

J.5.8 Extended bit-field types

1 A bit-field may be declared with a type other than **`_Bool`**, **`unsigned int`**, or **`signed int`**, with an appropriate maximum width (6.7.2.1).

J.5.9 The `fortran` keyword

1 The **`fortran`** function specifier may be used in a function declaration to indicate that calls suitable for FORTRAN should be generated, or that a different representation for the external name is to be generated (6.7.4).

J.5.10 The `asm` keyword

1 The **`asm`** keyword may be used to insert assembly language directly into the translator output (6.8). The most common implementation is via a statement of the form:

`asm (` *character-string-literal* **`);`**

J.5.11 Multiple external definitions

1 There may be more than one external definition for the identifier of an object, with or without the explicit use of the keyword **`extern`**; if the definitions disagree, or more than one is initialized, the behavior is undefined (6.9.2).

J.5.12 Predefined macro names

1 Macro names that do not begin with an underscore, describing the translation and execution environments, are defined by the implementation before translation begins (6.10.8).

J.5.13 Floating-point status flags

1 If any floating-point status flags are set on normal termination after all calls to functions registered by the **`atexit`** function have been made (see 7.22.4.4), the implementation writes some diagnostics indicating the fact to the **`stderr`** stream, if it is still open,

J.5.14 Extra arguments for signal handlers

1 Handlers for specific signals are called with extra arguments in addition to the signal number (7.14.1.1).

J.5.15 Additional stream types and file-opening modes

1 Additional mappings from files to streams are supported (7.21.2).

2 Additional file-opening modes may be specified by characters appended to the **`mode`** argument of the **`fopen`** function (7.21.5.3).

J.5.16 Defined file position indicator

1 The file position indicator is decremented by each successful call to the **`ungetc`** or **`ungetwc`** function for a text stream, except if its value was zero before a call (7.21.7.10, 7.29.3.10).

J.5.17 Math error reporting

1 Functions declared in **`<complex.h>`** and **`<math.h>`** raise **`SIGFPE`** to report errors instead of, or in addition to, setting **`errno`** or raising floating-point exceptions (7.3, 7.12).

Annex K
(normative)
Bounds-checking interfaces

K.1 Background

1 Traditionally, the C Library has contained many functions that trust the programmer to provide output character arrays big enough to hold the result being produced. Not only do these functions not check that the arrays are big enough, they frequently lack the information needed to perform such checks. While it is possible to write safe, robust, and error-free code using the existing library, the library tends to promote programming styles that lead to mysterious failures if a result is too big for the provided array.

2 A common programming style is to declare character arrays large enough to handle most practical cases. However, if these arrays are not large enough to handle the resulting strings, data can be written past the end of the array overwriting other data and program structures. The program never gets any indication that a problem exists, and so never has a chance to recover or to fail gracefully.

3 Worse, this style of programming has compromised the security of computers and networks. Buffer overflows can often be exploited to run arbitrary code with the permissions of the vulnerable (defective) program.

4 If the programmer writes runtime checks to verify lengths before calling library functions, then those runtime checks frequently duplicate work done inside the library functions, which discover string lengths as a side effect of doing their job.

5 This annex provides alternative library functions that promote safer, more secure programming. The alternative functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not. Data is never written past the end of an array. All string results are null terminated.

6 This annex also addresses another problem that complicates writing robust code: functions that are not reentrant because they return pointers to static objects owned by the function. Such functions can be troublesome since a previously returned result can change if the function is called again, perhaps by another thread.

K.2 Scope

1 This annex specifies a series of optional extensions that can be useful in the mitigation of security vulnerabilities in programs, and comprise new functions, macros, and types declared or defined in existing standard headers.

2 An implementation that defines **`__STDC_LIB_EXT1__`** shall conform to the specifications in this annex.[380)]

3 Subclause K.3 should be read as if it were merged into the parallel structure of named subclauses of clause 7.

K.3 Library

K.3.1 Introduction

K.3.1.1 Standard headers

1 The functions, macros, and types declared or defined in K.3 and its subclauses are not declared or defined by their respective headers if **`__STDC_WANT_LIB_EXT1__`** is defined as a macro which expands to the integer constant **`0`** at the point in the source file where the appropriate header is first included.

2 The functions, macros, and types declared or defined in K.3 and its subclauses are declared and defined by their respective headers if **`__STDC_WANT_LIB_EXT1__`** is defined as a macro which expands to the integer constant **`1`** at the point in the source file where the appropriate header is first included.[381)]

3 It is implementation-defined whether the functions, macros, and types declared or defined in K.3 and its subclauses are declared or defined by their respective headers if **`__STDC_WANT_LIB_EXT1__`** is not defined as a macro at the point in the source file where the appropriate header is first included.[382)]

4 Within a preprocessing translation unit, **`__STDC_WANT_LIB_EXT1__`** shall be defined identically for all inclusions of any headers from subclause K.3. If **`__STDC_WANT_LIB_EXT1__`** is defined differently for any such inclusion, the implementation shall issue a diagnostic as if a preprocessor error directive were used.

---

380) Implementations that do not define **`__STDC_LIB_EXT1__`** are not required to conform to these specifications.

381) Future revisions of this International Standard may define meanings for other values of **`__STDC_WANT_LIB_EXT1__`**.

382) Subclause 7.1.3 reserves certain names and patterns of names that an implementation may use in headers. All other names are not reserved, and a conforming implementation is not permitted to use them. While some of the names defined in K.3 and its subclauses are reserved, others are not. If an unreserved name is defined in a header when **`__STDC_WANT_LIB_EXT1__`** is defined as **`0`**, the implementation is not conforming.

K.3.1.2 Reserved identifiers

1 Each macro name in any of the following subclauses is reserved for use as specified if it is defined by any of its associated headers when included; unless explicitly stated otherwise (see 7.1.4).

2 All identifiers with external linkage in any of the following subclauses are reserved for use as identifiers with external linkage if any of them are used by the program. None of them are reserved if none of them are used.

3 Each identifier with file scope listed in any of the following subclauses is reserved for use as a macro name and as an identifier with file scope in the same name space if it is defined by any of its associated headers when included.

K.3.1.3 Use of errno

1 An implementation may set **errno** for the functions defined in this annex, but is not required to.

K.3.1.4 Runtime-constraint violations

1 Most functions in this annex include as part of their specification a list of runtime-constraints. These runtime-constraints are requirements on the program using the library.[383)]

2 Implementations shall verify that the runtime-constraints for a function are not violated by the program. If a runtime-constraint is violated, the implementation shall call the currently registered runtime-constraint handler (see **set_constraint_handler_s** in **<stdlib.h>**). Multiple runtime-constraint violations in the same call to a library function result in only one call to the runtime-constraint handler. It is unspecified which one of the multiple runtime-constraint violations cause the handler to be called.

3 If the runtime-constraints section for a function states an action to be performed when a runtime-constraint violation occurs, the function shall perform the action before calling the runtime-constraint handler. If the runtime-constraints section lists actions that are prohibited when a runtime-constraint violation occurs, then such actions are prohibited to the function both before calling the handler and after the handler returns.

4 The runtime-constraint handler might not return. If the handler does return, the library function whose runtime-constraint was violated shall return some indication of failure as given by the returns section in the function's specification.

383) Although runtime-constraints replace many cases of undefined behavior, undefined behavior still exists in this annex. Implementations are free to detect any case of undefined behavior and treat it as a runtime-constraint violation by calling the runtime-constraint handler. This license comes directly from the definition of undefined behavior.

K.3.2 Errors `<errno.h>`

1 The header `<errno.h>` defines a type.

2 The type is

```
errno_t
```

which is type `int`.[384)]

K.3.3 Common definitions `<stddef.h>`

1 The header `<stddef.h>` defines a type.

2 The type is

```
rsize_t
```

which is the type `size_t`.[385)]

K.3.4 Integer types `<stdint.h>`

1 The header `<stdint.h>` defines a macro.

2 The macro is

```
RSIZE_MAX
```

which expands to a value[386)] of type `size_t`. Functions that have parameters of type `rsize_t` consider it a runtime-constraint violation if the values of those parameters are greater than `RSIZE_MAX`.

Recommended practice

3 Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like `size_t`. Also, some implementations do not support objects as large as the maximum value that can be represented by type `size_t`.

4 For those reasons, it is sometimes beneficial to restrict the range of object sizes to detect programming errors. For implementations targeting machines with large address spaces, it is recommended that `RSIZE_MAX` be defined as the smaller of the size of the largest object supported or `(SIZE_MAX >> 1)`, even if this limit is smaller than the size of some legitimate, but very large, objects. Implementations targeting machines with small address spaces may wish to define `RSIZE_MAX` as `SIZE_MAX`, which means that there

---

384) As a matter of programming style, `errno_t` may be used as the type of something that deals only with the values that might be found in `errno`. For example, a function which returns the value of `errno` might be declared as having the return type `errno_t`.

385) See the description of the `RSIZE_MAX` macro in `<stdint.h>`.

386) The macro `RSIZE_MAX` need not expand to a constant expression.

is no object size that is considered a runtime-constraint violation.

K.3.5 Input/output `<stdio.h>`

1 The header **`<stdio.h>`** defines several macros and two types.

2 The macros are

```
L_tmpnam_s
```

which expands to an integer constant expression that is the size needed for an array of **`char`** large enough to hold a temporary file name string generated by the **`tmpnam_s`** function;

```
TMP_MAX_S
```

which expands to an integer constant expression that is the maximum number of unique file names that can be generated by the **`tmpnam_s`** function.

3 The types are

```
errno_t
```

which is type **`int`**; and

```
rsize_t
```

which is the type **`size_t`**.

K.3.5.1 Operations on files

K.3.5.1.1 The `tmpfile_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t tmpfile_s(FILE * restrict * restrict streamptr);
```

Runtime-constraints

2 **`streamptr`** shall not be a null pointer.

3 If there is a runtime-constraint violation, **`tmpfile_s`** does not attempt to create a file.

Description

4 The **`tmpfile_s`** function creates a temporary binary file that is different from any other existing file and that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with **`"wb+"`** mode with the meaning that mode has in the **`fopen_s`** function (including the mode's effect on exclusive access and file permissions).

5 If the file was created successfully, then the pointer to **FILE** pointed to by **streamptr** will be set to the pointer to the object controlling the opened file. Otherwise, the pointer to **FILE** pointed to by **streamptr** will be set to a null pointer.

Recommended practice

It should be possible to open at least **TMP_MAX_S** temporary files during the lifetime of the program (this limit may be shared with **tmpnam_s**) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files (**FOPEN_MAX**).

Returns

6 The **tmpfile_s** function returns zero if it created the file. If it did not create the file or there was a runtime-constraint violation, **tmpfile_s** returns a nonzero value.

K.3.5.1.2 The tmpnam_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t tmpnam_s(char *s, rsize_t maxsize);
```

Runtime-constraints

2 **s** shall not be a null pointer. **maxsize** shall be less than or equal to **RSIZE_MAX**. **maxsize** shall be greater than the length of the generated file name string.

Description

3 The **tmpnam_s** function generates a string that is a valid file name and that is not the same as the name of an existing file.[387] The function is potentially capable of generating **TMP_MAX_S** different strings, but any or all of them may already be in use by existing files and thus not be suitable return values. The lengths of these strings shall be less than the value of the **L_tmpnam_s** macro.

4 The **tmpnam_s** function generates a different string each time it is called.

5 It is assumed that **s** points to an array of at least **maxsize** characters. This array will be set to generated string, as specified below.

387) Files created using strings generated by the **tmpnam_s** function are temporary only in the sense that their names should not collide with those generated by conventional naming rules for the implementation. It is still necessary to use the **remove** function to remove such files when their use is ended, and before program termination. Implementations should take care in choosing the patterns used for names returned by **tmpnam_s**. For example, making a thread id part of the names avoids the race condition and possible conflict when multiple programs run simultaneously by the same user generate the same temporary file names.

6 The implementation shall behave as if no library function except **tmpnam** calls the **tmpnam_s** function.[388)]

Recommended practice

7 After a program obtains a file name using the **tmpnam_s** function and before the program creates a file with that name, the possibility exists that someone else may create a file with that same name. To avoid this race condition, the **tmpfile_s** function should be used instead of **tmpnam_s** when possible. One situation that requires the use of the **tmpnam_s** function is when the program needs to create a temporary directory rather than a temporary file.

Returns

8 If no suitable string can be generated, or if there is a runtime-constraint violation, the **tmpnam_s** function writes a null character to **s[0]** (only if **s** is not null and **maxsize** is greater than zero) and returns a nonzero value.

9 Otherwise, the **tmpnam_s** function writes the string in the array pointed to by **s** and returns zero.

Environmental limits

10 The value of the macro **TMP_MAX_S** shall be at least 25.

K.3.5.2 File access functions

K.3.5.2.1 The fopen_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t fopen_s(FILE * restrict * restrict streamptr,
     const char * restrict filename,
     const char * restrict mode);
```

Runtime-constraints

2 None of **streamptr**, **filename**, or **mode** shall be a null pointer.

3 If there is a runtime-constraint violation, **fopen_s** does not attempt to open a file. Furthermore, if **streamptr** is not a null pointer, **fopen_s** sets ***streamptr** to the null pointer.

388) An implementation may have **tmpnam** call **tmpnam_s** (perhaps so there is only one naming convention for temporary files), but this is not required.

Description

4 The **fopen_s** function opens the file whose name is the string pointed to by **filename**, and associates a stream with it.

5 The **mode** string shall be as described for **fopen**, with the addition that modes starting with the character **'w'** or **'a'** may be preceded by the character **'u'**, see below:

uw	truncate to zero length or create text file for writing, default permissions
uwx	create text file for writing, default permissions
ua	append; open or create text file for writing at end-of-file, default permissions
uwb	truncate to zero length or create binary file for writing, default permissions
uwbx	create binary file for writing, default permissions
uab	append; open or create binary file for writing at end-of-file, default permissions
uw+	truncate to zero length or create text file for update, default permissions
uw+x	create text file for update, default permissions
ua+	append; open or create text file for update, writing at end-of-file, default permissions
uw+b *or* **uwb+**	truncate to zero length or create binary file for update, default permissions
uw+bx *or* **uwb+x**	create binary file for update, default permissions
ua+b *or* **uab+**	append; open or create binary file for update, writing at end-of-file, default permissions

6 Opening a file with exclusive mode (**'x'** as the last character in the **mode** argument) fails if the file already exists or cannot be created.

7 To the extent that the underlying system supports the concepts, files opened for writing shall be opened with exclusive (also known as non-shared) access. If the file is being created, and the first character of the mode string is not **'u'**, to the extent that the underlying system supports it, the file shall have a file permission that prevents other users on the system from accessing the file. If the file is being created and first character of the mode string is **'u'**, then by the time the file has been closed, it shall have the system default file access permissions.[389)]

8 If the file was opened successfully, then the pointer to **FILE** pointed to by **streamptr** will be set to the pointer to the object controlling the opened file. Otherwise, the pointer

389) These are the same permissions that the file would have been created with by **fopen**.

to **FILE** pointed to by **streamptr** will be set to a null pointer.

Returns

9 The **fopen_s** function returns zero if it opened the file. If it did not open the file or if there was a runtime-constraint violation, **fopen_s** returns a nonzero value.

K.3.5.2.2 The freopen_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t freopen_s(FILE * restrict * restrict newstreamptr,
     const char * restrict filename,
     const char * restrict mode,
     FILE * restrict stream);
```

Runtime-constraints

2 None of **newstreamptr**, **mode**, and **stream** shall be a null pointer.

3 If there is a runtime-constraint violation, **freopen_s** neither attempts to close any file associated with **stream** nor attempts to open a file. Furthermore, if **newstreamptr** is not a null pointer, **fopen_s** sets ***newstreamptr** to the null pointer.

Description

4 The **freopen_s** function opens the file whose name is the string pointed to by **filename** and associates the stream pointed to by **stream** with it. The **mode** argument has the same meaning as in the **fopen_s** function (including the mode's effect on exclusive access and file permissions).

5 If **filename** is a null pointer, the **freopen_s** function attempts to change the mode of the stream to that specified by **mode**, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

6 The **freopen_s** function first attempts to close any file that is associated with **stream**. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared.

7 If the file was opened successfully, then the pointer to **FILE** pointed to by **newstreamptr** will be set to the value of **stream**. Otherwise, the pointer to **FILE** pointed to by **newstreamptr** will be set to a null pointer.

Returns

8 The **freopen_s** function returns zero if it opened the file. If it did not open the file or there was a runtime-constraint violation, **freopen_s** returns a nonzero value.

K.3.5.3 Formatted input/output functions

1 Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the objects take on unspecified values.

K.3.5.3.1 The `fprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int fprintf_s(FILE * restrict stream,
     const char * restrict format, ...);
```

Runtime-constraints

2 Neither **`stream`** nor **`format`** shall be a null pointer. The **`%n`** specifier[390] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by **`format`**. Any argument to **`fprintf_s`** corresponding to a **`%s`** specifier shall not be a null pointer.

3 If there is a runtime-constraint violation,[391] the **`fprintf_s`** function does not attempt to produce further output, and it is unspecified to what extent **`fprintf_s`** produced output before discovering the runtime-constraint violation.

Description

4 The **`fprintf_s`** function is equivalent to the **`fprintf`** function except for the explicit runtime-constraints listed above.

Returns

5 The **`fprintf_s`** function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

390) It is not a runtime-constraint violation for the characters **`%n`** to appear in sequence in the string pointed at by **`format`** when those characters are not a interpreted as a **`%n`** specifier. For example, if the entire format string was **`%%n`**.

391) Because an implementation may treat any undefined behavior as a runtime-constraint violation, an implementation may treat any unsupported specifiers in the string pointed to by **`format`** as a runtime-constraint violation.

K.3.5.3.2 The `fscanf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int fscanf_s(FILE * restrict stream,
     const char * restrict format, ...);
```

Runtime-constraints

2 Neither **`stream`** nor **`format`** shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation,[392)] the **`fscanf_s`** function does not attempt to perform further input, and it is unspecified to what extent **`fscanf_s`** performed input before discovering the runtime-constraint violation.

Description

4 The **`fscanf_s`** function is equivalent to **`fscanf`** except that the **`c`**, **`s`**, and **`[`** conversion specifiers apply to a pair of arguments (unless assignment suppression is indicated by a **`*`**). The first of these arguments is the same as for **`fscanf`**. That argument is immediately followed in the argument list by the second argument, which has type **`rsize_t`** and gives the number of elements in the array pointed to by the first argument of the pair. If the first argument points to a scalar object, it is considered to be an array of one element.[393)]

5 A matching failure occurs if the number of elements in a receiving object is insufficient to hold the converted input (including any trailing null character).

Returns

6 The **`fscanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the

---

392) Because an implementation may treat any undefined behavior as a runtime-constraint violation, an implementation may treat any unsupported specifiers in the string pointed to by **`format`** as a runtime-constraint violation.

393) If the format is known at translation time, an implementation may issue a diagnostic for any argument used to store the result from a **`c`**, **`s`**, or **`[`** conversion specifier if that argument is not followed by an argument of a type compatible with **`rsize_t`**. A limited amount of checking may be done if even if the format is not known at translation time. For example, an implementation may issue a diagnostic for each argument after **`format`** that has of type pointer to one of **`char`**, **`signed char`**, **`unsigned char`**, or **`void`** that is not followed by an argument of a type compatible with **`rsize_t`**. The diagnostic could warn that unless the pointer is being used with a conversion specifier using the **`hh`** length modifier, a length argument must follow the pointer argument. Another useful diagnostic could flag any non-pointer argument following **`format`** that did not have a type compatible with **`rsize_t`**.

fscanf_s function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7 EXAMPLE 1 The call:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
/* ... */
int n, i; float x; char name[50];
n = fscanf_s(stdin, "%d%f%s", &i, &x, name, (rsize_t) 50);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to **n** the value 3, to **i** the value 25, to **x** the value 5.432, and to **name** the sequence **thompson\0**.

8 EXAMPLE 2 The call:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
/* ... */
int n; char s[5];
n = fscanf_s(stdin, "%s", s, sizeof s);
```

with the input line:

```
hello
```

will assign to **n** the value 0 since a matching failure occurred because the sequence **hello\0** requires an array of six characters to store it.

K.3.5.3.3 The printf_s function

Synopsis

1

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int printf_s(const char * restrict format, ...);
```

Runtime-constraints

2 **format** shall not be a null pointer. The **%n** specifier[394] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by **format**. Any argument to **printf_s** corresponding to a **%s** specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **printf_s** function does not attempt to produce further output, and it is unspecified to what extent **printf_s** produced output before discovering the runtime-constraint violation.

394) It is not a runtime-constraint violation for the characters **%n** to appear in sequence in the string pointed at by **format** when those characters are not a interpreted as a **%n** specifier. For example, if the entire format string was **%%n**.

Description

4 The **printf_s** function is equivalent to the **printf** function except for the explicit runtime-constraints listed above.

Returns

5 The **printf_s** function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.5.3.4 The scanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int scanf_s(const char * restrict format, ...);
```

Runtime-constraints

2 **format** shall not be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **scanf_s** function does not attempt to perform further input, and it is unspecified to what extent **scanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **scanf_s** function is equivalent to **fscanf_s** with the argument **stdin** interposed before the arguments to **scanf_s**.

Returns

5 The **scanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **scanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.3.5 The snprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int snprintf_s(char * restrict s, rsize_t n,
     const char * restrict format, ...);
```

Runtime-constraints

2 Neither **s** nor **format** shall be a null pointer. **n** shall neither equal zero nor be greater than **RSIZE_MAX**. The **%n** specifier[395] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by **format**. Any argument to

snprintf_s corresponding to a **%s** specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if **s** is not a null pointer and **n** is greater than zero and less than **RSIZE_MAX**, then the **snprintf_s** function sets **s[0]** to the null character.

Description

4 The **snprintf_s** function is equivalent to the **snprintf** function except for the explicit runtime-constraints listed above.

5 The **snprintf_s** function, unlike **sprintf_s**, will truncate the result to fit within the array pointed to by **s**.

Returns

6 The **snprintf_s** function returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if a runtime-constraint violation occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

K.3.5.3.6 The sprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int sprintf_s(char * restrict s, rsize_t n,
     const char * restrict format, ...);
```

Runtime-constraints

2 Neither **s** nor **format** shall be a null pointer. **n** shall neither equal zero nor be greater than **RSIZE_MAX**. The number of characters (including the trailing null) required for the result to be written to the array pointed to by **s** shall not be greater than **n**. The **%n** specifier[396] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by **format**. Any argument to **sprintf_s** corresponding to a **%s** specifier shall not be a null pointer. No encoding error shall occur.

---

395) It is not a runtime-constraint violation for the characters **%n** to appear in sequence in the string pointed at by **format** when those characters are not a interpreted as a **%n** specifier. For example, if the entire format string was **%%n**.

396) It is not a runtime-constraint violation for the characters **%n** to appear in sequence in the string pointed at by **format** when those characters are not a interpreted as a **%n** specifier. For example, if the entire format string was **%%n**.

3 If there is a runtime-constraint violation, then if **s** is not a null pointer and **n** is greater than zero and less than **RSIZE_MAX**, then the **sprintf_s** function sets **s[0]** to the null character.

Description

4 The **sprintf_s** function is equivalent to the **sprintf** function except for the parameter **n** and the explicit runtime-constraints listed above.

5 The **sprintf_s** function, unlike **snprintf_s**, treats a result too big for the array pointed to by **s** as a runtime-constraint violation.

Returns

6 If no runtime-constraint violation occurred, the **sprintf_s** function returns the number of characters written in the array, not counting the terminating null character. If an encoding error occurred, **sprintf_s** returns a negative value. If any other runtime-constraint violation occurred, **sprintf_s** returns zero.

K.3.5.3.7 The sscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int sscanf_s(const char * restrict s,
     const char * restrict format, ...);
```

Runtime-constraints

2 Neither **s** nor **format** shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **sscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **sscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **sscanf_s** function is equivalent to **fscanf_s**, except that input is obtained from a string (specified by the argument **s**) rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the **fscanf_s** function. If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

5 The **sscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **sscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.3.8 The `vfprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vfprintf_s(FILE * restrict stream,
     const char * restrict format,
     va_list arg);
```

Runtime-constraints

2 Neither **`stream`** nor **`format`** shall be a null pointer. The **`%n`** specifier[397] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by **`format`**. Any argument to **`vfprintf_s`** corresponding to a **`%s`** specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **`vfprintf_s`** function does not attempt to produce further output, and it is unspecified to what extent **`vfprintf_s`** produced output before discovering the runtime-constraint violation.

Description

4 The **`vfprintf_s`** function is equivalent to the **`vfprintf`** function except for the explicit runtime-constraints listed above.

Returns

5 The **`vfprintf_s`** function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.5.3.9 The `vfscanf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vfscanf_s(FILE * restrict stream,
     const char * restrict format,
     va_list arg);
```

397) It is not a runtime-constraint violation for the characters **`%n`** to appear in sequence in the string pointed at by **`format`** when those characters are not a interpreted as a **`%n`** specifier. For example, if the entire format string was **`%%n`**.

Runtime-constraints

2 Neither **stream** nor **format** shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vfscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **vfscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **vfscanf_s** function is equivalent to **fscanf_s**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfscanf_s** function does not invoke the **va_end** macro.[398)]

Returns

5 The **vfscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **vfscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.3.10 The vprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vprintf_s(const char * restrict format,
     va_list arg);
```

Runtime-constraints

2 **format** shall not be a null pointer. The **%n** specifier[399)] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by **format**. Any argument to **vprintf_s** corresponding to a **%s** specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vprintf_s** function does not attempt to produce further output, and it is unspecified to what extent **vprintf_s** produced output before discovering the runtime-constraint violation.

---

398) As the functions **vfprintf_s**, **vfscanf_s**, **vprintf_s**, **vscanf_s**, **vsnprintf_s**, **vsprintf_s**, and **vsscanf_s** invoke the **va_arg** macro, the value of **arg** after the return is indeterminate.

399) It is not a runtime-constraint violation for the characters **%n** to appear in sequence in the string pointed at by **format** when those characters are not a interpreted as a **%n** specifier. For example, if the entire format string was **%%n**.

Description

4 The **vprintf_s** function is equivalent to the **vprintf** function except for the explicit runtime-constraints listed above.

Returns

5 The **vprintf_s** function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.5.3.11 The vscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vscanf_s(const char * restrict format,
     va_list arg);
```

Runtime-constraints

2 **format** shall not be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **vscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **vscanf_s** function is equivalent to **scanf_s**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vscanf_s** function does not invoke the **va_end** macro.[400)]

Returns

5 The **vscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **vscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

400) As the functions **vfprintf_s**, **vfscanf_s**, **vprintf_s**, **vscanf_s**, **vsnprintf_s**, **vsprintf_s**, and **vsscanf_s** invoke the **va_arg** macro, the value of **arg** after the return is indeterminate.

K.3.5.3.12 The vsnprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vsnprintf_s(char * restrict s, rsize_t n,
     const char * restrict format,
     va_list arg);
```

Runtime-constraints

2 Neither **s** nor **format** shall be a null pointer. **n** shall neither equal zero nor be greater than **RSIZE_MAX**. The **%n** specifier[401] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by **format**. Any argument to **vsnprintf_s** corresponding to a **%s** specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if **s** is not a null pointer and **n** is greater than zero and less than **RSIZE_MAX**, then the **vsnprintf_s** function sets **s[0]** to the null character.

Description

4 The **vsnprintf_s** function is equivalent to the **vsnprintf** function except for the explicit runtime-constraints listed above.

5 The **vsnprintf_s** function, unlike **vsprintf_s**, will truncate the result to fit within the array pointed to by **s**.

Returns

6 The **vsnprintf_s** function returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if a runtime-constraint violation occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

---

401) It is not a runtime-constraint violation for the characters **%n** to appear in sequence in the string pointed at by **format** when those characters are not a interpreted as a **%n** specifier. For example, if the entire format string was **%%n**.

K.3.5.3.13 The `vsprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vsprintf_s(char * restrict s, rsize_t n,
     const char * restrict format,
     va_list arg);
```

Runtime-constraints

2 Neither **s** nor **format** shall be a null pointer. **n** shall neither equal zero nor be greater than **RSIZE_MAX**. The number of characters (including the trailing null) required for the result to be written to the array pointed to by **s** shall not be greater than **n**. The **%n** specifier[402] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by **format**. Any argument to **vsprintf_s** corresponding to a **%s** specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if **s** is not a null pointer and **n** is greater than zero and less than **RSIZE_MAX**, then the **vsprintf_s** function sets **s[0]** to the null character.

Description

4 The **vsprintf_s** function is equivalent to the **vsprintf** function except for the parameter **n** and the explicit runtime-constraints listed above.

5 The **vsprintf_s** function, unlike **vsnprintf_s**, treats a result too big for the array pointed to by **s** as a runtime-constraint violation.

Returns

6 If no runtime-constraint violation occurred, the **vsprintf_s** function returns the number of characters written in the array, not counting the terminating null character. If an encoding error occurred, **vsprintf_s** returns a negative value. If any other runtime-constraint violation occurred, **vsprintf_s** returns zero.

402) It is not a runtime-constraint violation for the characters **%n** to appear in sequence in the string pointed at by **format** when those characters are not a interpreted as a **%n** specifier. For example, if the entire format string was **%%n**.

K.3.5.3.14 The `vsscanf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vsscanf_s(const char * restrict s,
     const char * restrict format,
     va_list arg);
```

Runtime-constraints

2 Neither **`s`** nor **`format`** shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **`vsscanf_s`** function does not attempt to perform further input, and it is unspecified to what extent **`vsscanf_s`** performed input before discovering the runtime-constraint violation.

Description

4 The **`vsscanf_s`** function is equivalent to **`sscanf_s`**, with the variable argument list replaced by **`arg`**, which shall have been initialized by the **`va_start`** macro (and possibly subsequent **`va_arg`** calls). The **`vsscanf_s`** function does not invoke the **`va_end`** macro.[403)]

Returns

5 The **`vsscanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **`vscanf_s`** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.4 Character input/output functions

K.3.5.4.1 The `gets_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
char *gets_s(char *s, rsize_t n);
```

403) As the functions **`vfprintf_s`**, **`vfscanf_s`**, **`vprintf_s`**, **`vscanf_s`**, **`vsnprintf_s`**, **`vsprintf_s`**, and **`vsscanf_s`** invoke the **`va_arg`** macro, the value of **`arg`** after the return is indeterminate.

Runtime-constraints

2 **s** shall not be a null pointer. **n** shall neither be equal to zero nor be greater than **RSIZE_MAX**. A new-line character, end-of-file, or read error shall occur within reading **n-1** characters from **stdin**.[404)]

3 If there is a runtime-constraint violation, **s[0]** is set to the null character, and characters are read and discarded from **stdin** until a new-line character is read, or end-of-file or a read error occurs.

Description

4 The **gets_s** function reads at most one less than the number of characters specified by **n** from the stream pointed to by **stdin**, into the array pointed to by **s**. No additional characters are read after a new-line character (which is discarded) or after end-of-file. The discarded new-line character does not count towards number of characters read. A null character is written immediately after the last character read into the array.

5 If end-of-file is encountered and no characters have been read into the array, or if a read error occurs during the operation, then **s[0]** is set to the null character, and the other elements of **s** take unspecified values.

Recommended practice

6 The **fgets** function allows properly-written programs to safely process input lines too long to store in the result array. In general this requires that callers of **fgets** pay attention to the presence or absence of a new-line character in the result array. Consider using **fgets** (along with any needed processing based on new-line characters) instead of **gets_s**.

Returns

7 The **gets_s** function returns **s** if successful. If there was a runtime-constraint violation, or if end-of-file is encountered and no characters have been read into the array, or if a read error occurs during the operation, then a null pointer is returned.

---

404) The **gets_s** function, unlike the historical **gets** function, makes it a runtime-constraint violation for a line of input to overflow the buffer to store it. Unlike the **fgets** function, **gets_s** maintains a one-to-one relationship between input lines and successful calls to **gets_s**. Programs that use **gets** expect such a relationship.

K.3.6 General utilities `<stdlib.h>`

1 The header `<stdlib.h>` defines three types.

2 The types are

```
errno_t
```

which is type `int`; and

```
rsize_t
```

which is the type `size_t`; and

```
constraint_handler_t
```

which has the following definition

```
typedef void (*constraint_handler_t)(
     const char * restrict msg,
     void * restrict ptr,
     errno_t error);
```

K.3.6.1 Runtime-constraint handling

K.3.6.1.1 The `set_constraint_handler_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
constraint_handler_t set_constraint_handler_s(
     constraint_handler_t handler);
```

Description

2 The `set_constraint_handler_s` function sets the runtime-constraint handler to be `handler`. The runtime-constraint handler is the function to be called when a library function detects a runtime-constraint violation. Only the most recent handler registered with `set_constraint_handler_s` is called when a runtime-constraint violation occurs.

3 When the handler is called, it is passed the following arguments in the following order:

1. A pointer to a character string describing the runtime-constraint violation.
2. A null pointer or a pointer to an implementation defined object.
3. If the function calling the handler has a return type declared as `errno_t`, the return value of the function is passed. Otherwise, a positive value of type `errno_t` is passed.

4 The implementation has a default constraint handler that is used if no calls to the **set_constraint_handler_s** function have been made. The behavior of the default handler is implementation-defined, and it may cause the program to exit or abort.

5 If the **handler** argument to **set_constraint_handler_s** is a null pointer, the implementation default handler becomes the current constraint handler.

Returns

6 The **set_constraint_handler_s** function returns a pointer to the previously registered handler.[405)]

K.3.6.1.2 The abort_handler_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
void abort_handler_s(
     const char * restrict msg,
     void * restrict ptr,
     errno_t error);
```

Description

2 A pointer to the **abort_handler_s** function shall be a suitable argument to the **set_constraint_handler_s** function.

3 The **abort_handler_s** function writes a message on the standard error stream in an implementation-defined format. The message shall include the string pointed to by **msg**. The **abort_handler_s** function then calls the **abort** function.[406)]

Returns

4 The **abort_handler_s** function does not return to its caller.

---

405) If the previous handler was registered by calling **set_constraint_handler_s** with a null pointer argument, a pointer to the implementation default handler is returned (not NULL).

406) Many implementations invoke a debugger when the **abort** function is called.

K.3.6.1.3 The `ignore_handler_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
void ignore_handler_s(
     const char * restrict msg,
     void * restrict ptr,
     errno_t error);
```

Description

2 A pointer to the **`ignore_handler_s`** function shall be a suitable argument to the **`set_constraint_handler_s`** function.

3 The **`ignore_handler_s`** function simply returns to its caller.[407)]

Returns

4 The **`ignore_handler_s`** function returns no value.

K.3.6.2 Communication with the environment

K.3.6.2.1 The `getenv_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
errno_t getenv_s(size_t * restrict len,
          char * restrict value, rsize_t maxsize,
          const char * restrict name);
```

Runtime-constraints

2 **`name`** shall not be a null pointer. **`maxsize`** shall neither equal zero nor be greater than **`RSIZE_MAX`**. If **`maxsize`** is not equal to zero, then **`value`** shall not be a null pointer.

3 If there is a runtime-constraint violation, the integer pointed to by **`len`** is set to 0 (if **`len`** is not null), and the environment list is not searched.

Description

4 The **`getenv_s`** function searches an *environment list*, provided by the host environment, for a string that matches the string pointed to by **`name`**.

---

407) If the runtime-constraint handler is set to the **`ignore_handler_s`** function, any library function in which a runtime-constraint violation occurs will return to its caller. The caller can determine whether a runtime-constraint violation occurred based on the library function's specification (usually, the library function returns a nonzero **`errno_t`**).

5 If that name is found then **getenv_s** performs the following actions. If **len** is not a null pointer, the length of the string associated with the matched list member is stored in the integer pointed to by **len**. If the length of the associated string is less than **maxsize**, then the associated string is copied to the array pointed to by **value**.

6 If that name is not found then **getenv_s** performs the following actions. If **len** is not a null pointer, zero is stored in the integer pointed to by **len**. If **maxsize** is greater than zero, then **value[0]** is set to the null character.

7 The set of environment names and the method for altering the environment list are implementation-defined. The **getenv_s** function need not avoid data races with other threads of execution that modify the environment list.[408)]

Returns

8 The **getenv_s** function returns zero if the specified **name** is found and the associated string was successfully stored in **value**. Otherwise, a nonzero value is returned.

K.3.6.3 Searching and sorting utilities

1 These utilities make use of a comparison function to search or sort arrays of unspecified type. Where an argument declared as **size_t nmemb** specifies the length of the array for a function, if **nmemb** has the value zero on a call to that function, then the comparison function is not called, a search finds no matching element, sorting performs no rearrangement, and the pointer to the array may be null.

2 The implementation shall ensure that the second argument of the comparison function (when called from **bsearch_s**), or both arguments (when called from **qsort_s**), are pointers to elements of the array.[409)] The first argument when called from **bsearch_s** shall equal **key**.

3 The comparison function shall not alter the contents of either the array or search key. The implementation may reorder elements of the array between calls to the comparison function, but shall not otherwise alter the contents of any individual element.

4 When the same objects (consisting of **size** bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, for **qsort_s** they shall define a total ordering on the array, and for **bsearch_s** the same object shall always compare the same way with the key.

---

408) Many implementations provide non-standard functions that modify the environment list.

409) That is, if the value passed is **p**, then the following expressions are always valid and nonzero:

```
((char *)p - (char *)base) % size == 0
(char *)p >= (char *)base
(char *)p < (char *)base + nmemb * size
```

5 A sequence point occurs immediately before and immediately after each call to the comparison function, and also between any call to the comparison function and any movement of the objects passed as arguments to that call.

K.3.6.3.1 The `bsearch_s` function

Synopsis

1

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
void *bsearch_s(const void *key, const void *base,
     rsize_t nmemb, rsize_t size,
     int (*compar)(const void *k, const void *y,
                    void *context),
     void *context);
```

Runtime-constraints

2 Neither **nmemb** nor **size** shall be greater than **RSIZE_MAX**. If **nmemb** is not equal to zero, then none of **key**, **base**, or **compar** shall be a null pointer.

3 If there is a runtime-constraint violation, the **bsearch_s** function does not search the array.

Description

4 The **bsearch_s** function searches an array of **nmemb** objects, the initial element of which is pointed to by **base**, for an element that matches the object pointed to by **key**. The size of each element of the array is specified by **size**.

5 The comparison function pointed to by **compar** is called with three arguments. The first two point to the **key** object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the **key** object is considered, respectively, to be less than, to match, or to be greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the **key** object, in that order.[410] The third argument to the comparison function is the **context** argument passed to **bsearch_s**. The sole use of **context** by **bsearch_s** is to pass it to the comparison function.[411]

410) In practice, this means that the entire array has been sorted according to the comparison function.

411) The **context** argument is for the use of the comparison function in performing its duties. For example, it might specify a collating sequence used by the comparison function.

Returns

6 The **`bsearch_s`** function returns a pointer to a matching element of the array, or a null pointer if no match is found or there is a runtime-constraint violation. If two elements compare as equal, which element is matched is unspecified.

K.3.6.3.2 The `qsort_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
errno_t qsort_s(void *base, rsize_t nmemb, rsize_t size,
     int (*compar)(const void *x, const void *y,
                    void *context),
     void *context);
```

Runtime-constraints

2 Neither **`nmemb`** nor **`size`** shall be greater than **`RSIZE_MAX`**. If **`nmemb`** is not equal to zero, then neither **`base`** nor **`compar`** shall be a null pointer.

3 If there is a runtime-constraint violation, the **`qsort_s`** function does not sort the array.

Description

4 The **`qsort_s`** function sorts an array of **`nmemb`** objects, the initial element of which is pointed to by **`base`**. The size of each object is specified by **`size`**.

5 The contents of the array are sorted into ascending order according to a comparison function pointed to by **`compar`**, which is called with three arguments. The first two point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. The third argument to the comparison function is the **`context`** argument passed to **`qsort_s`**. The sole use of **`context`** by **`qsort_s`** is to pass it to the comparison function.[412)]

6 If two elements compare as equal, their relative order in the resulting sorted array is unspecified.

Returns

7 The **`qsort_s`** function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

412) The **`context`** argument is for the use of the comparison function in performing its duties. For example, it might specify a collating sequence used by the comparison function.

K.3.6.4 Multibyte/wide character conversion functions

1 The behavior of the multibyte character functions is affected by the **LC_CTYPE** category of the current locale. For a state-dependent encoding, each function is placed into its initial conversion state by a call for which its character pointer argument, **s**, is a null pointer. Subsequent calls with **s** as other than a null pointer cause the internal conversion state of the function to be altered as necessary. A call with **s** as a null pointer causes these functions to set the int pointed to by their **status** argument to a nonzero value if encodings have state dependency, and zero otherwise.[413] Changing the **LC_CTYPE** category causes the conversion state of these functions to be indeterminate.

K.3.6.4.1 The wctomb_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
errno_t wctomb_s(int * restrict status,
     char * restrict s,
     rsize_t smax,
     wchar_t wc);
```

Runtime-constraints

2 Let *n* denote the number of bytes needed to represent the multibyte character corresponding to the wide character given by **wc** (including any shift sequences).

3 If **s** is not a null pointer, then **smax** shall not be less than *n*, and **smax** shall not be greater than **RSIZE_MAX**. If **s** is a null pointer, then **smax** shall equal zero.

4 If there is a runtime-constraint violation, **wctomb_s** does not modify the int pointed to by **status**, and if **s** is not a null pointer, no more than **smax** elements in the array pointed to by **s** will be accessed.

Description

5 The **wctomb_s** function determines *n* and stores the multibyte character representation of **wc** in the array whose first element is pointed to by **s** (if **s** is not a null pointer). The number of characters stored never exceeds **MB_CUR_MAX** or **smax**. If **wc** is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state, and the function is left in the initial conversion state.

6 The implementation shall behave as if no library function calls the **wctomb_s** function.

413) If the locale employs special bytes to change the shift state, these bytes do not produce separate wide character codes, but are grouped with an adjacent multibyte character.

7 If **s** is a null pointer, the **wctomb_s** function stores into the int pointed to by **status** a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings.

8 If **s** is not a null pointer, the **wctomb_s** function stores into the int pointed to by **status** either *n* or −1 if **wc**, respectively, does or does not correspond to a valid multibyte character.

9 In no case will the int pointed to by **status** be set to a value greater than the **MB_CUR_MAX** macro.

Returns

10 The **wctomb_s** function returns zero if successful, and a nonzero value if there was a runtime-constraint violation or **wc** did not correspond to a valid multibyte character.

K.3.6.5 Multibyte/wide string conversion functions

1 The behavior of the multibyte string functions is affected by the **LC_CTYPE** category of the current locale.

K.3.6.5.1 The mbstowcs_s function

Synopsis

1
```
#include <stdlib.h>
errno_t mbstowcs_s(size_t * restrict retval,
     wchar_t * restrict dst, rsize_t dstmax,
     const char * restrict src, rsize_t len);
```

Runtime-constraints

2 Neither **retval** nor **src** shall be a null pointer. If **dst** is not a null pointer, then neither **len** nor **dstmax** shall be greater than **RSIZE_MAX**. If **dst** is a null pointer, then **dstmax** shall equal zero. If **dst** is not a null pointer, then **dstmax** shall not equal zero. If **dst** is not a null pointer and **len** is not less than **dstmax**, then a null character shall occur within the first **dstmax** multibyte characters of the array pointed to by **src**.

3 If there is a runtime-constraint violation, then **mbstowcs_s** does the following. If **retval** is not a null pointer, then **mbstowcs_s** sets ***retval** to **(size_t)(-1)**. If **dst** is not a null pointer and **dstmax** is greater than zero and less than **RSIZE_MAX**, then **mbstowcs_s** sets **dst[0]** to the null wide character.

Description

4 The **mbstowcs_s** function converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by **src** into a sequence of corresponding wide characters. If **dst** is not a null pointer, the converted characters are stored into the array pointed to by **dst**. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of

bytes is encountered that does not form a valid multibyte character, or (if `dst` is not a null pointer) when `len` wide characters have been stored into the array pointed to by `dst`.[414] If `dst` is not a null pointer and no null wide character was stored into the array pointed to by `dst`, then `dst[len]` is set to the null wide character. Each conversion takes place as if by a call to the `mbrtowc` function.

5 Regardless of whether `dst` is or is not a null pointer, if the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the `mbstowcs_s` function stores the value `(size_t)(-1)` into `*retval`. Otherwise, the `mbstowcs_s` function stores into `*retval` the number of multibyte characters successfully converted, not including the terminating null character (if any).

6 All elements following the terminating null wide character (if any) written by `mbstowcs_s` in the array of `dstmax` wide characters pointed to by `dst` take unspecified values when `mbstowcs_s` returns.[415]

7 If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

8 The `mbstowcs_s` function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

K.3.6.5.2 The `wcstombs_s` function

Synopsis

1
```
#include <stdlib.h>
errno_t wcstombs_s(size_t * restrict retval,
     char * restrict dst, rsize_t dstmax,
     const wchar_t * restrict src, rsize_t len);
```

Runtime-constraints

2 Neither `retval` nor `src` shall be a null pointer. If `dst` is not a null pointer, then neither `len` nor `dstmax` shall be greater than `RSIZE_MAX`. If `dst` is a null pointer, then `dstmax` shall equal zero. If `dst` is not a null pointer, then `dstmax` shall not equal zero. If `dst` is not a null pointer and `len` is not less than `dstmax`, then the conversion shall have been stopped (see below) because a terminating null wide character was reached or because an encoding error occurred.

---

414) Thus, the value of `len` is ignored if `dst` is a null pointer.

415) This allows an implementation to attempt converting the multibyte string before discovering a terminating null character did not occur where required.

3 If there is a runtime-constraint violation, then **wcstombs_s** does the following. If **retval** is not a null pointer, then **wcstombs_s** sets ***retval** to **(size_t)(-1)**. If **dst** is not a null pointer and **dstmax** is greater than zero and less than **RSIZE_MAX**, then **wcstombs_s** sets **dst[0]** to the null character.

Description

4 The **wcstombs_s** function converts a sequence of wide characters from the array pointed to by **src** into a sequence of corresponding multibyte characters that begins in the initial shift state. If **dst** is not a null pointer, the converted characters are then stored into the array pointed to by **dst**. Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier in two cases:

— when a wide character is reached that does not correspond to a valid multibyte character;

— (if **dst** is not a null pointer) when the next multibyte character would exceed the limit of *n* total bytes to be stored into the array pointed to by **dst**. If the wide character being converted is the null wide character, then *n* is the lesser of **len** or **dstmax**. Otherwise, *n* is the lesser of **len** or **dstmax-1**.

If the conversion stops without converting a null wide character and **dst** is not a null pointer, then a null character is stored into the array pointed to by **dst** immediately following any multibyte characters already stored. Each conversion takes place as if by a call to the **wcrtomb** function.[416)]

5 Regardless of whether **dst** is or is not a null pointer, if the input conversion encounters a wide character that does not correspond to a valid multibyte character, an encoding error occurs: the **wcstombs_s** function stores the value **(size_t)(-1)** into ***retval**. Otherwise, the **wcstombs_s** function stores into ***retval** the number of bytes in the resulting multibyte character sequence, not including the terminating null character (if any).

6 All elements following the terminating null character (if any) written by **wcstombs_s** in the array of **dstmax** elements pointed to by **dst** take unspecified values when **wcstombs_s** returns.[417)]

7 If copying takes place between objects that overlap, the objects take on unspecified values.

416) If conversion stops because a terminating null wide character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte. However, if the conversion stops before a terminating null wide character has been reached, the result will be null terminated, but might not end in the initial shift state.

417) When **len** is not less than **dstmax**, the implementation might fill the array before discovering a runtime-constraint violation.

Returns

8 The **wcstombs_s** function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

K.3.7 String handling **<string.h>**

1 The header **<string.h>** defines two types.

2 The types are

```
errno_t
```

which is type **int**; and

```
rsize_t
```

which is the type **size_t**.

K.3.7.1 Copying functions

K.3.7.1.1 The **memcpy_s** function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t memcpy_s(void * restrict s1, rsize_t s1max,
     const void * restrict s2, rsize_t n);
```

Runtime-constraints

2 Neither **s1** nor **s2** shall be a null pointer. Neither **s1max** nor **n** shall be greater than **RSIZE_MAX**. **n** shall not be greater than **s1max**. Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, the **memcpy_s** function stores zeros in the first **s1max** characters of the object pointed to by **s1** if **s1** is not a null pointer and **s1max** is not greater than **RSIZE_MAX**.

Description

4 The **memcpy_s** function copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**.

Returns

5 The **memcpy_s** function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.1.2 The `memmove_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t memmove_s(void *s1, rsize_t s1max,
     const void *s2, rsize_t n);
```

Runtime-constraints

2 Neither **s1** nor **s2** shall be a null pointer. Neither **s1max** nor **n** shall be greater than **RSIZE_MAX**. **n** shall not be greater than **s1max**.

3 If there is a runtime-constraint violation, the **memmove_s** function stores zeros in the first **s1max** characters of the object pointed to by **s1** if **s1** is not a null pointer and **s1max** is not greater than **RSIZE_MAX**.

Description

4 The **memmove_s** function copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. This copying takes place as if the **n** characters from the object pointed to by **s2** are first copied into a temporary array of **n** characters that does not overlap the objects pointed to by **s1** or **s2**, and then the **n** characters from the temporary array are copied into the object pointed to by **s1**.

Returns

5 The **memmove_s** function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.1.3 The `strcpy_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strcpy_s(char * restrict s1,
     rsize_t s1max,
     const char * restrict s2);
```

Runtime-constraints

2 Neither **s1** nor **s2** shall be a null pointer. **s1max** shall not be greater than **RSIZE_MAX**. **s1max** shall not equal zero. **s1max** shall be greater than **strnlen_s(s2, s1max)**. Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, then if **s1** is not a null pointer and **s1max** is greater than zero and not greater than **RSIZE_MAX**, then **strcpy_s** sets **s1[0]** to the null character.

Description

4 The **strcpy_s** function copies the string pointed to by **s2** (including the terminating null character) into the array pointed to by **s1**.

5 All elements following the terminating null character (if any) written by **strcpy_s** in the array of **s1max** characters pointed to by **s1** take unspecified values when **strcpy_s** returns.[418)]

Returns

6 The **strcpy_s** function returns zero[419)] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.1.4 The strncpy_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strncpy_s(char * restrict s1,
     rsize_t s1max,
     const char * restrict s2,
     rsize_t n);
```

Runtime-constraints

2 Neither **s1** nor **s2** shall be a null pointer. Neither **s1max** nor **n** shall be greater than **RSIZE_MAX**. **s1max** shall not equal zero. If **n** is not less than **s1max**, then **s1max** shall be greater than **strnlen_s(s2, s1max)**. Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, then if **s1** is not a null pointer and **s1max** is greater than zero and not greater than **RSIZE_MAX**, then **strncpy_s** sets **s1[0]** to the null character.

Description

4 The **strncpy_s** function copies not more than **n** successive characters (characters that follow a null character are not copied) from the array pointed to by **s2** to the array pointed to by **s1**. If no null character was copied from **s2**, then **s1[n]** is set to a null character.

418) This allows an implementation to copy characters from **s2** to **s1** while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of **s1** before discovering that the first element should be set to the null character.

419) A zero return value implies that all of the requested characters from the string pointed to by **s2** fit within the array pointed to by **s1** and that the result in **s1** is null terminated.

5 All elements following the terminating null character (if any) written by **strncpy_s** in the array of **s1max** characters pointed to by **s1** take unspecified values when **strncpy_s** returns.[420)]

Returns

6 The **strncpy_s** function returns zero[421)] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

7 EXAMPLE 1 The **strncpy_s** function can be used to copy a string without the danger that the result will not be null terminated or that characters will be written past the end of the destination array.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
/* ... */
char src1[100] = "hello";
char src2[7] = {'g', 'o', 'o', 'd', 'b', 'y', 'e'};
char dst1[6], dst2[5], dst3[5];
int r1, r2, r3;
r1 = strncpy_s(dst1, 6, src1, 100);
r2 = strncpy_s(dst2, 5, src2, 7);
r3 = strncpy_s(dst3, 5, src2, 4);
```

The first call will assign to **r1** the value zero and to **dst1** the sequence **hello\0**.
The second call will assign to **r2** a nonzero value and to **dst2** the sequence **\0**.
The third call will assign to **r3** the value zero and to **dst3** the sequence **good\0**.

K.3.7.2 Concatenation functions

K.3.7.2.1 The strcat_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strcat_s(char * restrict s1,
     rsize_t s1max,
     const char * restrict s2);
```

Runtime-constraints

2 Let *m* denote the value **s1max - strnlen_s(s1, s1max)** upon entry to **strcat_s**.

---

420) This allows an implementation to copy characters from **s2** to **s1** while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of **s1** before discovering that the first element should be set to the null character.

421) A zero return value implies that all of the requested characters from the string pointed to by **s2** fit within the array pointed to by **s1** and that the result in **s1** is null terminated.

3 Neither **s1** nor **s2** shall be a null pointer. **s1max** shall not be greater than **RSIZE_MAX**. **s1max** shall not equal zero. *m* shall not equal zero.[422] *m* shall be greater than **strnlen_s(s2,** *m***)**. Copying shall not take place between objects that overlap.

4 If there is a runtime-constraint violation, then if **s1** is not a null pointer and **s1max** is greater than zero and not greater than **RSIZE_MAX**, then **strcat_s** sets **s1[0]** to the null character.

Description

5 The **strcat_s** function appends a copy of the string pointed to by **s2** (including the terminating null character) to the end of the string pointed to by **s1**. The initial character from **s2** overwrites the null character at the end of **s1**.

6 All elements following the terminating null character (if any) written by **strcat_s** in the array of **s1max** characters pointed to by **s1** take unspecified values when **strcat_s** returns.[423]

Returns

7 The **strcat_s** function returns zero[424] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.2.2 The strncat_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strncat_s(char * restrict s1,
     rsize_t s1max,
     const char * restrict s2,
     rsize_t n);
```

Runtime-constraints

2 Let *m* denote the value **s1max - strnlen_s(s1, s1max)** upon entry to **strncat_s**.

3 Neither **s1** nor **s2** shall be a null pointer. Neither **s1max** nor **n** shall be greater than **RSIZE_MAX**. **s1max** shall not equal zero. *m* shall not equal zero.[425] If **n** is not less

422) Zero means that **s1** was not null terminated upon entry to **strcat_s**.

423) This allows an implementation to append characters from **s2** to **s1** while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of **s1** before discovering that the first element should be set to the null character.

424) A zero return value implies that all of the requested characters from the string pointed to by **s2** were appended to the string pointed to by **s1** and that the result in **s1** is null terminated.

than *m*, then *m* shall be greater than **`strnlen_s(s2, m)`**. Copying shall not take place between objects that overlap.

4 If there is a runtime-constraint violation, then if **`s1`** is not a null pointer and **`s1max`** is greater than zero and not greater than **`RSIZE_MAX`**, then **`strncat_s`** sets **`s1[0]`** to the null character.

Description

5 The **`strncat_s`** function appends not more than **`n`** successive characters (characters that follow a null character are not copied) from the array pointed to by **`s2`** to the end of the string pointed to by **`s1`**. The initial character from **`s2`** overwrites the null character at the end of **`s1`**. If no null character was copied from **`s2`**, then **`s1[s1max-`***m***`+n]`** is set to a null character.

6 All elements following the terminating null character (if any) written by **`strncat_s`** in the array of **`s1max`** characters pointed to by **`s1`** take unspecified values when **`strncat_s`** returns.[426)]

Returns

7 The **`strncat_s`** function returns zero[427)] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

8 EXAMPLE 1 The **`strncat_s`** function can be used to copy a string without the danger that the result will not be null terminated or that characters will be written past the end of the destination array.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
/* ... */
char s1[100] = "good";
char s2[6] = "hello";
char s3[6] = "hello";
char s4[7] = "abc";
char s5[1000] = "bye";
int r1, r2, r3, r4;
r1 = strncat_s(s1, 100, s5, 1000);
r2 = strncat_s(s2, 6, "", 1);
r3 = strncat_s(s3, 6, "X", 2);
r4 = strncat_s(s4, 7, "defghijklmn", 3);
```

After the first call **`r1`** will have the value zero and **`s1`** will contain the sequence **`goodbye\0`**.

425) Zero means that **`s1`** was not null terminated upon entry to **`strncat_s`**.

426) This allows an implementation to append characters from **`s2`** to **`s1`** while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of **`s1`** before discovering that the first element should be set to the null character.

427) A zero return value implies that all of the requested characters from the string pointed to by **`s2`** were appended to the string pointed to by **`s1`** and that the result in **`s1`** is null terminated.

After the second call **r2** will have the value zero and **s2** will contain the sequence **hello\0**.
After the third call **r3** will have a nonzero value and **s3** will contain the sequence **\0**.
After the fourth call **r4** will have the value zero and **s4** will contain the sequence **abcdef\0**.

K.3.7.3 Search functions

K.3.7.3.1 The strtok_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
char *strtok_s(char * restrict s1,
     rsize_t * restrict s1max,
     const char * restrict s2,
     char ** restrict ptr);
```

Runtime-constraints

2 None of **s1max**, **s2**, or **ptr** shall be a null pointer. If **s1** is a null pointer, then ***ptr** shall not be a null pointer. The value of ***s1max** shall not be greater than **RSIZE_MAX**. The end of the token found shall occur within the first ***s1max** characters of **s1** for the first call, and shall occur within the first ***s1max** characters of where searching resumes on subsequent calls.

3 If there is a runtime-constraint violation, the **strtok_s** function does not indirect through the **s1** or **s2** pointers, and does not store a value in the object pointed to by **ptr**.

Description

4 A sequence of calls to the **strtok_s** function breaks the string pointed to by **s1** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **s2**. The fourth argument points to a caller-provided **char** pointer into which the **strtok_s** function stores information necessary for it to continue scanning the same string.

5 The first call in a sequence has a non-null first argument and **s1max** points to an object whose value is the number of elements in the character array pointed to by the first argument. The first call stores an initial value in the object pointed to by **ptr** and updates the value pointed to by **s1max** to reflect the number of elements that remain in relation to **ptr**. Subsequent calls in the sequence have a null first argument and the objects pointed to by **s1max** and **ptr** are required to have the values stored by the previous call in the sequence, which are then updated. The separator string pointed to by **s2** may be different from call to call.

6 The first call in the sequence searches the string pointed to by **s1** for the first character that is *not* contained in the current separator string pointed to by **s2**. If no such character is found, then there are no tokens in the string pointed to by **s1** and the **strtok_s** function returns a null pointer. If such a character is found, it is the start of the first token.

7 The **strtok_s** function then searches from there for the first character in **s1** that *is* contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **s1**, and subsequent searches in the same string for a token return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token.

8 In all cases, the **strtok_s** function stores sufficient information in the pointer pointed to by **ptr** so that subsequent calls, with a null pointer for **s1** and the unmodified pointer value for **ptr**, shall start searching just past the element overwritten by a null character (if any).

Returns

9 The **strtok_s** function returns a pointer to the first character of a token, or a null pointer if there is no token or there is a runtime-constraint violation.

10 EXAMPLE

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
static char str1[] = "?a???b,,,#c";
static char str2[] = "\t \t";
char *t, *ptr1, *ptr2;
rsize_t max1 = sizeof (str1);
rsize_t max2 = sizeof (str2);

t = strtok_s(str1, &max1, "?", &ptr1);     // t points to the token "a"
t = strtok_s(NULL, &max1, ",", &ptr1);     // t points to the token "??b"
t = strtok_s(str2, &max2, " \t", &ptr2);   // t is a null pointer
t = strtok_s(NULL, &max1, "#,", &ptr1);    // t points to the token "c"
t = strtok_s(NULL, &max1, "?", &ptr1);     // t is a null pointer
```

K.3.7.4 Miscellaneous functions

K.3.7.4.1 The memset_s function

Synopsis

1

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t memset_s(void *s, rsize_t smax, int c, rsize_t n)
```

Runtime-constraints

2 **s** shall not be a null pointer. Neither **smax** nor **n** shall be greater than **RSIZE_MAX**. **n** shall not be greater than **smax**.

3 If there is a runtime-constraint violation, then if **s** is not a null pointer and **smax** is not greater than **RSIZE_MAX**, the **memset_s** function stores the value of **c** (converted to an **unsigned char**) into each of the first **smax** characters of the object pointed to by **s**.

Description

4 The **memset_s** function copies the value of **c** (converted to an **unsigned char**) into each of the first **n** characters of the object pointed to by **s**. Unlike **memset**, any call to the **memset_s** function shall be evaluated strictly according to the rules of the abstract machine as described in (5.1.2.3). That is, any call to the **memset_s** function shall assume that the memory indicated by **s** and **n** may be accessible in the future and thus must contain the values indicated by **c**.

Returns

5 The **memset_s** function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.4.2 The strerror_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strerror_s(char *s, rsize_t maxsize,
     errno_t errnum);
```

Runtime-constraints

2 **s** shall not be a null pointer. **maxsize** shall not be greater than **RSIZE_MAX**. **maxsize** shall not equal zero.

3 If there is a runtime-constraint violation, then the array (if any) pointed to by **s** is not modified.

Description

4 The **strerror_s** function maps the number in **errnum** to a locale-specific message string. Typically, the values for **errnum** come from **errno**, but **strerror_s** shall map any value of type **int** to a message.

5 If the length of the desired string is less than **maxsize**, then the string is copied to the array pointed to by **s**.

6 Otherwise, if **maxsize** is greater than zero, then **maxsize-1** characters are copied from the string to the array pointed to by **s** and then **s[maxsize-1]** is set to the null character. Then, if **maxsize** is greater than 3, then **s[maxsize-2]**, **s[maxsize-3]**, and **s[maxsize-4]** are set to the character period (**.**).

Returns

7 The **strerror_s** function returns zero if the length of the desired string was less than **maxsize** and there was no runtime-constraint violation. Otherwise, the **strerror_s** function returns a nonzero value.

K.3.7.4.3 The `strerrorlen_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
size_t strerrorlen_s(errno_t errnum);
```

Description

2 The **strerrorlen_s** function calculates the length of the (untruncated) locale-specific message string that the **strerror_s** function maps to **errnum**.

Returns

3 The **strerrorlen_s** function returns the number of characters (not including the null character) in the full message string.

K.3.7.4.4 The `strnlen_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
size_t strnlen_s(const char *s, size_t maxsize);
```

Description

2 The **strnlen_s** function computes the length of the string pointed to by **s**.

Returns

3 If **s** is a null pointer,[428)] then the **strnlen_s** function returns zero.

4 Otherwise, the **strnlen_s** function returns the number of characters that precede the terminating null character. If there is no null character in the first **maxsize** characters of **s** then **strnlen_s** returns **maxsize**. At most the first **maxsize** characters of **s** shall be accessed by **strnlen_s**.

428) Note that the **strnlen_s** function has no runtime-constraints. This lack of runtime-constraints along with the values returned for a null pointer or an unterminated string argument make **strnlen_s** useful in algorithms that gracefully handle such exceptional data.

K.3.8 Date and time `<time.h>`

1 The header **`<time.h>`** defines two types.

2 The types are

```
errno_t
```

which is type **`int`**; and

```
rsize_t
```

which is the type **`size_t`**.

K.3.8.1 Components of time

1 A broken-down time is *normalized* if the values of the members of the **`tm`** structure are in their normal rages.[429)]

K.3.8.2 Time conversion functions

1 Like the **`strftime`** function, the **`asctime_s`** and **`ctime_s`** functions do not return a pointer to a static object, and other library functions are permitted to call them.

K.3.8.2.1 The `asctime_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>
errno_t asctime_s(char *s, rsize_t maxsize,
     const struct tm *timeptr);
```

Runtime-constraints

2 Neither **`s`** nor **`timeptr`** shall be a null pointer. **`maxsize`** shall not be less than 26 and shall not be greater than **`RSIZE_MAX`**. The broken-down time pointed to by **`timeptr`** shall be normalized. The calendar year represented by the broken-down time pointed to by **`timeptr`** shall not be less than calendar year 0 and shall not be greater than calendar year 9999.

3 If there is a runtime-constraint violation, there is no attempt to convert the time, and **`s[0]`** is set to a null character if **`s`** is not a null pointer and **`maxsize`** is not zero and is not greater than **`RSIZE_MAX`**.

Description

4 The **`asctime_s`** function converts the normalized broken-down time in the structure pointed to by **`timeptr`** into a 26 character (including the null character) string in the

429) The normal ranges are defined in 7.27.1.

form

```
Sun Sep 16 01:03:52 1973\n\0
```

The fields making up this string are (in order):

1. The name of the day of the week represented by **`timeptr->tm_wday`** using the following three character weekday names: Sun, Mon, Tue, Wed, Thu, Fri, and Sat.
2. The character space.
3. The name of the month represented by **`timeptr->tm_mon`** using the following three character month names: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec.
4. The character space.
5. The value of **`timeptr->tm_mday`** as if printed using the **`fprintf`** format **`"%2d"`**.
6. The character space.
7. The value of **`timeptr->tm_hour`** as if printed using the **`fprintf`** format **`"%.2d"`**.
8. The character colon.
9. The value of **`timeptr->tm_min`** as if printed using the **`fprintf`** format **`"%.2d"`**.
10. The character colon.
11. The value of **`timeptr->tm_sec`** as if printed using the **`fprintf`** format **`"%.2d"`**.
12. The character space.
13. The value of **`timeptr->tm_year + 1900`** as if printed using the **`fprintf`** format **`"%4d"`**.
14. The character new line.
15. The null character.

Recommended practice

The **`strftime`** function allows more flexible formatting and supports locale-specific behavior. If you do not require the exact form of the result string produced by the **`asctime_s`** function, consider using the **`strftime`** function instead.

Returns

5 The **`asctime_s`** function returns zero if the time was successfully converted and stored into the array pointed to by **`s`**. Otherwise, it returns a nonzero value.

K.3.8.2.2 The `ctime_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>
errno_t ctime_s(char *s, rsize_t maxsize,
     const time_t *timer);
```

Runtime-constraints

2 Neither **s** nor **timer** shall be a null pointer. **maxsize** shall not be less than 26 and shall not be greater than **RSIZE_MAX**.

3 If there is a runtime-constraint violation, **s[0]** is set to a null character if **s** is not a null pointer and **maxsize** is not equal zero and is not greater than **RSIZE_MAX**.

Description

4 The **ctime_s** function converts the calendar time pointed to by **timer** to local time in the form of a string. It is equivalent to

```
asctime_s(s, maxsize, localtime_s(timer))
```

Recommended practice

The **strftime** function allows more flexible formatting and supports locale-specific behavior. If you do not require the exact form of the result string produced by the **ctime_s** function, consider using the **strftime** function instead.

Returns

5 The **ctime_s** function returns zero if the time was successfully converted and stored into the array pointed to by **s**. Otherwise, it returns a nonzero value.

K.3.8.2.3 The `gmtime_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>
struct tm *gmtime_s(const time_t * restrict timer,
     struct tm * restrict result);
```

Runtime-constraints

2 Neither **timer** nor **result** shall be a null pointer.

3 If there is a runtime-constraint violation, there is no attempt to convert the time.

Description

4 The **gmtime_s** function converts the calendar time pointed to by **timer** into a broken-down time, expressed as UTC. The broken-down time is stored in the structure pointed

to by **`result`**.

Returns

5 The **`gmtime_s`** function returns **`result`**, or a null pointer if the specified time cannot be converted to UTC or there is a runtime-constraint violation.

K.3.8.2.4 The `localtime_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>
struct tm *localtime_s(const time_t * restrict timer,
     struct tm * restrict result);
```

Runtime-constraints

2 Neither **`timer`** nor **`result`** shall be a null pointer.

3 If there is a runtime-constraint violation, there is no attempt to convert the time.

Description

4 The **`localtime_s`** function converts the calendar time pointed to by **`timer`** into a broken-down time, expressed as local time. The broken-down time is stored in the structure pointed to by **`result`**.

Returns

5 The **`localtime_s`** function returns **`result`**, or a null pointer if the specified time cannot be converted to local time or there is a runtime-constraint violation.

K.3.9 Extended multibyte and wide character utilities `<wchar.h>`

1 The header **`<wchar.h>`** defines two types.

2 The types are

```
errno_t
```

which is type **`int`**; and

```
rsize_t
```

which is the type **`size_t`**.

3 Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the objects take on unspecified values.

K.3.9.1 Formatted wide character input/output functions

K.3.9.1.1 The `fwprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int fwprintf_s(FILE * restrict stream,
     const wchar_t * restrict format, ...);
```

Runtime-constraints

2 Neither **`stream`** nor **`format`** shall be a null pointer. The **`%n`** specifier[430] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by **`format`**. Any argument to **`fwprintf_s`** corresponding to a **`%s`** specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **`fwprintf_s`** function does not attempt to produce further output, and it is unspecified to what extent **`fwprintf_s`** produced output before discovering the runtime-constraint violation.

Description

4 The **`fwprintf_s`** function is equivalent to the **`fwprintf`** function except for the explicit runtime-constraints listed above.

Returns

5 The **`fwprintf_s`** function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.9.1.2 The `fwscanf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
#include <wchar.h>
int fwscanf_s(FILE * restrict stream,
     const wchar_t * restrict format, ...);
```

Runtime-constraints

2 Neither **`stream`** nor **`format`** shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

---

430) It is not a runtime-constraint violation for the wide characters **`%n`** to appear in sequence in the wide string pointed at by **`format`** when those wide characters are not a interpreted as a **`%n`** specifier. For example, if the entire format string was **`L"%%n"`**.

3 If there is a runtime-constraint violation, the **fwscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **fwscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **fwscanf_s** function is equivalent to **fwscanf** except that the **c**, **s**, and **[** conversion specifiers apply to a pair of arguments (unless assignment suppression is indicated by a *****). The first of these arguments is the same as for **fwscanf**. That argument is immediately followed in the argument list by the second argument, which has type **size_t** and gives the number of elements in the array pointed to by the first argument of the pair. If the first argument points to a scalar object, it is considered to be an array of one element.[431)]

5 A matching failure occurs if the number of elements in a receiving object is insufficient to hold the converted input (including any trailing null character).

Returns

6 The **fwscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **fwscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.3 The `snwprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int snwprintf_s(wchar_t * restrict s,
     rsize_t n,
     const wchar_t * restrict format, ...);
```

Runtime-constraints

2 Neither **s** nor **format** shall be a null pointer. **n** shall neither equal zero nor be greater than **RSIZE_MAX**. The **%n** specifier[432)] (modified or not by flags, field width, or

431) If the format is known at translation time, an implementation may issue a diagnostic for any argument used to store the result from a **c**, **s**, or **[** conversion specifier if that argument is not followed by an argument of a type compatible with **rsize_t**. A limited amount of checking may be done if even if the format is not known at translation time. For example, an implementation may issue a diagnostic for each argument after **format** that has of type pointer to one of **char**, **signed char**, **unsigned char**, or **void** that is not followed by an argument of a type compatible with **rsize_t**. The diagnostic could warn that unless the pointer is being used with a conversion specifier using the **hh** length modifier, a length argument must follow the pointer argument. Another useful diagnostic could flag any non-pointer argument following **format** that did not have a type compatible with **rsize_t**.

precision) shall not appear in the wide string pointed to by **`format`**. Any argument to **`snwprintf_s`** corresponding to a **`%s`** specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if **`s`** is not a null pointer and **`n`** is greater than zero and less than **`RSIZE_MAX`**, then the **`snwprintf_s`** function sets **`s[0]`** to the null wide character.

Description

4 The **`snwprintf_s`** function is equivalent to the **`swprintf`** function except for the explicit runtime-constraints listed above.

5 The **`snwprintf_s`** function, unlike **`swprintf_s`**, will truncate the result to fit within the array pointed to by **`s`**.

Returns

6 The **`snwprintf_s`** function returns the number of wide characters that would have been written had **`n`** been sufficiently large, not counting the terminating wide null character, or a negative value if a runtime-constraint violation occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **`n`**.

K.3.9.1.4 The `swprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int swprintf_s(wchar_t * restrict s, rsize_t n,
     const wchar_t * restrict format, ...);
```

Runtime-constraints

2 Neither **`s`** nor **`format`** shall be a null pointer. **`n`** shall neither equal zero nor be greater than **`RSIZE_MAX`**. The number of wide characters (including the trailing null) required for the result to be written to the array pointed to by **`s`** shall not be greater than **`n`**. The **`%n`** specifier[433] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by **`format`**. Any argument to **`swprintf_s`** corresponding to a **`%s`** specifier shall not be a null pointer. No encoding error shall occur.

432) It is not a runtime-constraint violation for the wide characters **`%n`** to appear in sequence in the wide string pointed at by **`format`** when those wide characters are not a interpreted as a **`%n`** specifier. For example, if the entire format string was **`L"%%n"`**.

433) It is not a runtime-constraint violation for the wide characters **`%n`** to appear in sequence in the wide string pointed at by **`format`** when those wide characters are not a interpreted as a **`%n`** specifier. For example, if the entire format string was **`L"%%n"`**.

3 If there is a runtime-constraint violation, then if **s** is not a null pointer and **n** is greater than zero and less than **RSIZE_MAX**, then the **swprintf_s** function sets **s[0]** to the null wide character.

Description

4 The **swprintf_s** function is equivalent to the **swprintf** function except for the explicit runtime-constraints listed above.

5 The **swprintf_s** function, unlike **snwprintf_s**, treats a result too big for the array pointed to by **s** as a runtime-constraint violation.

Returns

6 If no runtime-constraint violation occurred, the **swprintf_s** function returns the number of wide characters written in the array, not counting the terminating null wide character. If an encoding error occurred or if **n** or more wide characters are requested to be written, **swprintf_s** returns a negative value. If any other runtime-constraint violation occurred, **swprintf_s** returns zero.

K.3.9.1.5 The swscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int swscanf_s(const wchar_t * restrict s,
     const wchar_t * restrict format, ...);
```

Runtime-constraints

2 Neither **s** nor **format** shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **swscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **swscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **swscanf_s** function is equivalent to **fwscanf_s**, except that the argument **s** specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for the **fwscanf_s** function.

Returns

5 The **swscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **swscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.6 The `vfwprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf_s(FILE * restrict stream,
     const wchar_t * restrict format,
     va_list arg);
```

Runtime-constraints

2 Neither **`stream`** nor **`format`** shall be a null pointer. The **`%n`** specifier[434] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by **`format`**. Any argument to **`vfwprintf_s`** corresponding to a **`%s`** specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **`vfwprintf_s`** function does not attempt to produce further output, and it is unspecified to what extent **`vfwprintf_s`** produced output before discovering the runtime-constraint violation.

Description

4 The **`vfwprintf_s`** function is equivalent to the **`vfwprintf`** function except for the explicit runtime-constraints listed above.

Returns

5 The **`vfwprintf_s`** function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.9.1.7 The `vfwscanf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwscanf_s(FILE * restrict stream,
     const wchar_t * restrict format, va_list arg);
```

434) It is not a runtime-constraint violation for the wide characters **`%n`** to appear in sequence in the wide string pointed at by **`format`** when those wide characters are not a interpreted as a **`%n`** specifier. For example, if the entire format string was **`L"%%n"`**.

Runtime-constraints

2 Neither **stream** nor **format** shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vfwscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **vfwscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **vfwscanf_s** function is equivalent to **fwscanf_s**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfwscanf_s** function does not invoke the **va_end** macro.[435)]

Returns

5 The **vfwscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **vfwscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.8 The vsnwprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vsnwprintf_s(wchar_t * restrict s,
     rsize_t n,
     const wchar_t * restrict format,
     va_list arg);
```

Runtime-constraints

2 Neither **s** nor **format** shall be a null pointer. **n** shall neither equal zero nor be greater than **RSIZE_MAX**. The **%n** specifier[436)] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by **format**. Any argument to **vsnwprintf_s** corresponding to a **%s** specifier shall not be a null pointer. No encoding error shall occur.

---

435) As the functions **vfwscanf_s**, **vwscanf_s**, and **vswscanf_s** invoke the **va_arg** macro, the value of **arg** after the return is indeterminate.

436) It is not a runtime-constraint violation for the wide characters **%n** to appear in sequence in the wide string pointed at by **format** when those wide characters are not a interpreted as a **%n** specifier. For example, if the entire format string was **L"%%n"**.

3 If there is a runtime-constraint violation, then if **s** is not a null pointer and **n** is greater than zero and less than **RSIZE_MAX**, then the **vsnwprintf_s** function sets **s[0]** to the null wide character.

Description

4 The **vsnwprintf_s** function is equivalent to the **vswprintf** function except for the explicit runtime-constraints listed above.

5 The **vsnwprintf_s** function, unlike **vswprintf_s**, will truncate the result to fit within the array pointed to by **s**.

Returns

6 The **vsnwprintf_s** function returns the number of wide characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if a runtime-constraint violation occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

K.3.9.1.9 The vswprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vswprintf_s(wchar_t * restrict s,
     rsize_t n,
     const wchar_t * restrict format,
     va_list arg);
```

Runtime-constraints

2 Neither **s** nor **format** shall be a null pointer. **n** shall neither equal zero nor be greater than **RSIZE_MAX**. The number of wide characters (including the trailing null) required for the result to be written to the array pointed to by **s** shall not be greater than **n**. The **%n** specifier[437)] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by **format**. Any argument to **vswprintf_s** corresponding to a **%s** specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if **s** is not a null pointer and **n** is greater than zero and less than **RSIZE_MAX**, then the **vswprintf_s** function sets **s[0]** to the null wide character.

437) It is not a runtime-constraint violation for the wide characters **%n** to appear in sequence in the wide string pointed at by **format** when those wide characters are not a interpreted as a **%n** specifier. For example, if the entire format string was **L"%%n"**.

Description

4 The **vswprintf_s** function is equivalent to the **vswprintf** function except for the explicit runtime-constraints listed above.

5 The **vswprintf_s** function, unlike **vsnwprintf_s**, treats a result too big for the array pointed to by **s** as a runtime-constraint violation.

Returns

6 If no runtime-constraint violation occurred, the **vswprintf_s** function returns the number of wide characters written in the array, not counting the terminating null wide character. If an encoding error occurred or if **n** or more wide characters are requested to be written, **vswprintf_s** returns a negative value. If any other runtime-constraint violation occurred, **vswprintf_s** returns zero.

K.3.9.1.10 The vswscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vswscanf_s(const wchar_t * restrict s,
     const wchar_t * restrict format,
     va_list arg);
```

Runtime-constraints

2 Neither **s** nor **format** shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vswscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **vswscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **vswscanf_s** function is equivalent to **swscanf_s**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vswscanf_s** function does not invoke the **va_end** macro.[438)]

438) As the functions **vfwscanf_s**, **vwscanf_s**, and **vswscanf_s** invoke the **va_arg** macro, the value of **arg** after the return is indeterminate.

Returns

5 The **vswscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **vswscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.11 The vwprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vwprintf_s(const wchar_t * restrict format,
     va_list arg);
```

Runtime-constraints

2 **format** shall not be a null pointer. The **%n** specifier[439] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by **format**. Any argument to **vwprintf_s** corresponding to a **%s** specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vwprintf_s** function does not attempt to produce further output, and it is unspecified to what extent **vwprintf_s** produced output before discovering the runtime-constraint violation.

Description

4 The **vwprintf_s** function is equivalent to the **vwprintf** function except for the explicit runtime-constraints listed above.

Returns

5 The **vwprintf_s** function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

439) It is not a runtime-constraint violation for the wide characters **%n** to appear in sequence in the wide string pointed at by **format** when those wide characters are not a interpreted as a **%n** specifier. For example, if the entire format string was **L"%%n"**.

K.3.9.1.12 The `vwscanf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vwscanf_s(const wchar_t * restrict format,
     va_list arg);
```

Runtime-constraints

2 **`format`** shall not be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **`vwscanf_s`** function does not attempt to perform further input, and it is unspecified to what extent **`vwscanf_s`** performed input before discovering the runtime-constraint violation.

Description

4 The **`vwscanf_s`** function is equivalent to **`wscanf_s`**, with the variable argument list replaced by **`arg`**, which shall have been initialized by the **`va_start`** macro (and possibly subsequent **`va_arg`** calls). The **`vwscanf_s`** function does not invoke the **`va_end`** macro.[440)]

Returns

5 The **`vwscanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **`vwscanf_s`** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.13 The `wprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int wprintf_s(const wchar_t * restrict format, ...);
```

Runtime-constraints

2 **`format`** shall not be a null pointer. The **`%n`** specifier[441)] (modified or not by flags, field

---

440) As the functions **`vfwscanf_s`**, **`vwscanf_s`**, and **`vswscanf_s`** invoke the **`va_arg`** macro, the value of **`arg`** after the return is indeterminate.

441) It is not a runtime-constraint violation for the wide characters **`%n`** to appear in sequence in the wide string pointed at by **`format`** when those wide characters are not a interpreted as a **`%n`** specifier. For example, if the entire format string was **`L"%%n"`**.

width, or precision) shall not appear in the wide string pointed to by **format**. Any argument to **wprintf_s** corresponding to a **%s** specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **wprintf_s** function does not attempt to produce further output, and it is unspecified to what extent **wprintf_s** produced output before discovering the runtime-constraint violation.

Description

4 The **wprintf_s** function is equivalent to the **wprintf** function except for the explicit runtime-constraints listed above.

Returns

5 The **wprintf_s** function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.9.1.14 The wscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int wscanf_s(const wchar_t * restrict format, ...);
```

Runtime-constraints

2 **format** shall not be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **wscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **wscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **wscanf_s** function is equivalent to **fwscanf_s** with the argument **stdin** interposed before the arguments to **wscanf_s**.

Returns

5 The **wscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **wscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.2 General wide string utilities

K.3.9.2.1 Wide string copying functions

K.3.9.2.1.1 The `wcscpy_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wcscpy_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2);
```

Runtime-constraints

2 Neither **`s1`** nor **`s2`** shall be a null pointer. **`s1max`** shall not be greater than **`RSIZE_MAX`**. **`s1max`** shall not equal zero. **`s1max`** shall be greater than **`wcsnlen_s(s2, s1max)`**. Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, then if **`s1`** is not a null pointer and **`s1max`** is greater than zero and not greater than **`RSIZE_MAX`**, then **`wcscpy_s`** sets **`s1[0]`** to the null wide character.

Description

4 The **`wcscpy_s`** function copies the wide string pointed to by **`s2`** (including the terminating null wide character) into the array pointed to by **`s1`**.

5 All elements following the terminating null wide character (if any) written by **`wcscpy_s`** in the array of **`s1max`** wide characters pointed to by **`s1`** take unspecified values when **`wcscpy_s`** returns.[442)]

Returns

6 The **`wcscpy_s`** function returns zero[443)] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

---

442) This allows an implementation to copy wide characters from **`s2`** to **`s1`** while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of **`s1`** before discovering that the first element should be set to the null wide character.

443) A zero return value implies that all of the requested wide characters from the string pointed to by **`s2`** fit within the array pointed to by **`s1`** and that the result in **`s1`** is null terminated.

K.3.9.2.1.2 The `wcsncpy_s` function

Synopsis

7
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wcsncpy_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2,
     rsize_t n);
```

Runtime-constraints

8 Neither **s1** nor **s2** shall be a null pointer. Neither **s1max** nor **n** shall be greater than **RSIZE_MAX**. **s1max** shall not equal zero. If **n** is not less than **s1max**, then **s1max** shall be greater than **wcsnlen_s(s2, s1max)**. Copying shall not take place between objects that overlap.

9 If there is a runtime-constraint violation, then if **s1** is not a null pointer and **s1max** is greater than zero and not greater than **RSIZE_MAX**, then **wcsncpy_s** sets **s1[0]** to the null wide character.

Description

10 The **wcsncpy_s** function copies not more than **n** successive wide characters (wide characters that follow a null wide character are not copied) from the array pointed to by **s2** to the array pointed to by **s1**. If no null wide character was copied from **s2**, then **s1[n]** is set to a null wide character.

11 All elements following the terminating null wide character (if any) written by **wcsncpy_s** in the array of **s1max** wide characters pointed to by **s1** take unspecified values when **wcsncpy_s** returns.[444)]

Returns

12 The **wcsncpy_s** function returns zero[445)] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

13 EXAMPLE 1 The **wcsncpy_s** function can be used to copy a wide string without the danger that the result will not be null terminated or that wide characters will be written past the end of the destination array.

---

444) This allows an implementation to copy wide characters from **s2** to **s1** while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of **s1** before discovering that the first element should be set to the null wide character.

445) A zero return value implies that all of the requested wide characters from the string pointed to by **s2** fit within the array pointed to by **s1** and that the result in **s1** is null terminated.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
/* ... */
wchar_t src1[100] = L"hello";
wchar_t src2[7] = {L'g', L'o', L'o', L'd', L'b', L'y', L'e'};
wchar_t dst1[6], dst2[5], dst3[5];
int r1, r2, r3;
r1 = wcsncpy_s(dst1, 6, src1, 100);
r2 = wcsncpy_s(dst2, 5, src2, 7);
r3 = wcsncpy_s(dst3, 5, src2, 4);
```

The first call will assign to **r1** the value zero and to **dst1** the sequence of wide characters **hello\0**. The second call will assign to **r2** a nonzero value and to **dst2** the sequence of wide characters **\0**. The third call will assign to **r3** the value zero and to **dst3** the sequence of wide characters **good\0**.

K.3.9.2.1.3 The wmemcpy_s function

Synopsis

14

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wmemcpy_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2,
     rsize_t n);
```

Runtime-constraints

15 Neither **s1** nor **s2** shall be a null pointer. Neither **s1max** nor **n** shall be greater than **RSIZE_MAX**. **n** shall not be greater than **s1max**. Copying shall not take place between objects that overlap.

16 If there is a runtime-constraint violation, the **wmemcpy_s** function stores zeros in the first **s1max** wide characters of the object pointed to by **s1** if **s1** is not a null pointer and **s1max** is not greater than **RSIZE_MAX**.

Description

17 The **wmemcpy_s** function copies **n** successive wide characters from the object pointed to by **s2** into the object pointed to by **s1**.

Returns

18 The **wmemcpy_s** function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.9.2.1.4 The `wmemmove_s` function

Synopsis

19
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wmemmove_s(wchar_t *s1, rsize_t s1max,
     const wchar_t *s2, rsize_t n);
```

Runtime-constraints

20 Neither **`s1`** nor **`s2`** shall be a null pointer. Neither **`s1max`** nor **`n`** shall be greater than **`RSIZE_MAX`**. **`n`** shall not be greater than **`s1max`**.

21 If there is a runtime-constraint violation, the **`wmemmove_s`** function stores zeros in the first **`s1max`** wide characters of the object pointed to by **`s1`** if **`s1`** is not a null pointer and **`s1max`** is not greater than **`RSIZE_MAX`**.

Description

22 The **`wmemmove_s`** function copies **`n`** successive wide characters from the object pointed to by **`s2`** into the object pointed to by **`s1`**. This copying takes place as if the **`n`** wide characters from the object pointed to by **`s2`** are first copied into a temporary array of **`n`** wide characters that does not overlap the objects pointed to by **`s1`** or **`s2`**, and then the **`n`** wide characters from the temporary array are copied into the object pointed to by **`s1`**.

Returns

23 The **`wmemmove_s`** function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.9.2.2 Wide string concatenation functions

K.3.9.2.2.1 The `wcscat_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wcscat_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2);
```

Runtime-constraints

2 Let *m* denote the value **`s1max - wcsnlen_s(s1, s1max)`** upon entry to **`wcscat_s`**.

3 Neither **`s1`** nor **`s2`** shall be a null pointer. **`s1max`** shall not be greater than **`RSIZE_MAX`**. **`s1max`** shall not equal zero. *m* shall not equal zero.[446] *m* shall be greater than **`wcsnlen_s(s2,`** *m*). Copying shall not take place between objects that overlap.

4 If there is a runtime-constraint violation, then if **s1** is not a null pointer and **s1max** is greater than zero and not greater than **RSIZE_MAX**, then **wcscat_s** sets **s1[0]** to the null wide character.

Description

5 The **wcscat_s** function appends a copy of the wide string pointed to by **s2** (including the terminating null wide character) to the end of the wide string pointed to by **s1**. The initial wide character from **s2** overwrites the null wide character at the end of **s1**.

6 All elements following the terminating null wide character (if any) written by **wcscat_s** in the array of **s1max** wide characters pointed to by **s1** take unspecified values when **wcscat_s** returns.[447)]

Returns

7 The **wcscat_s** function returns zero[448)] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.9.2.2.2 The wcsncat_s function

Synopsis

8
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wcsncat_s(wchar_t * restrict s1,
     rsize_t s1max,
     const wchar_t * restrict s2,
     rsize_t n);
```

Runtime-constraints

9 Let *m* denote the value **s1max - wcsnlen_s(s1, s1max)** upon entry to **wcsncat_s**.

10 Neither **s1** nor **s2** shall be a null pointer. Neither **s1max** nor **n** shall be greater than **RSIZE_MAX**. **s1max** shall not equal zero. *m* shall not equal zero.[449)] If **n** is not less than *m*, then *m* shall be greater than **wcsnlen_s(s2,** ***m*****)**. Copying shall not take place between objects that overlap.

446) Zero means that **s1** was not null terminated upon entry to **wcscat_s**.

447) This allows an implementation to append wide characters from **s2** to **s1** while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of **s1** before discovering that the first element should be set to the null wide character.

448) A zero return value implies that all of the requested wide characters from the wide string pointed to by **s2** were appended to the wide string pointed to by **s1** and that the result in **s1** is null terminated.

449) Zero means that **s1** was not null terminated upon entry to **wcsncat_s**.

11 If there is a runtime-constraint violation, then if **s1** is not a null pointer and **s1max** is greater than zero and not greater than **RSIZE_MAX**, then **wcsncat_s** sets **s1[0]** to the null wide character.

Description

12 The **wcsncat_s** function appends not more than **n** successive wide characters (wide characters that follow a null wide character are not copied) from the array pointed to by **s2** to the end of the wide string pointed to by **s1**. The initial wide character from **s2** overwrites the null wide character at the end of **s1**. If no null wide character was copied from **s2**, then **s1[s1max-***m***+n]** is set to a null wide character.

13 All elements following the terminating null wide character (if any) written by **wcsncat_s** in the array of **s1max** wide characters pointed to by **s1** take unspecified values when **wcsncat_s** returns.[450)]

Returns

14 The **wcsncat_s** function returns zero[451)] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

15 EXAMPLE 1 The **wcsncat_s** function can be used to copy a wide string without the danger that the result will not be null terminated or that wide characters will be written past the end of the destination array.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
/* ... */
wchar_t s1[100] = L"good";
wchar_t s2[6] = L"hello";
wchar_t s3[6] = L"hello";
wchar_t s4[7] = L"abc";
wchar_t s5[1000] = L"bye";
int r1, r2, r3, r4;
r1 = wcsncat_s(s1, 100, s5, 1000);
r2 = wcsncat_s(s2, 6, L"", 1);
r3 = wcsncat_s(s3, 6, L"X", 2);
r4 = wcsncat_s(s4, 7, L"defghijklmn", 3);
```

After the first call **r1** will have the value zero and **s1** will be the wide character sequence **goodbye\0**.
After the second call **r2** will have the value zero and **s2** will be the wide character sequence **hello\0**.
After the third call **r3** will have a nonzero value and **s3** will be the wide character sequence **\0**.
After the fourth call **r4** will have the value zero and **s4** will be the wide character sequence **abcdef\0**.

450) This allows an implementation to append wide characters from **s2** to **s1** while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of **s1** before discovering that the first element should be set to the null wide character.

451) A zero return value implies that all of the requested wide characters from the wide string pointed to by **s2** were appended to the wide string pointed to by **s1** and that the result in **s1** is null terminated.

K.3.9.2.3 Wide string search functions

K.3.9.2.3.1 The `wcstok_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
wchar_t *wcstok_s(wchar_t * restrict s1,
     rsize_t * restrict s1max,
     const wchar_t * restrict s2,
     wchar_t ** restrict ptr);
```

Runtime-constraints

2 None of **`s1max`**, **`s2`**, or **`ptr`** shall be a null pointer. If **`s1`** is a null pointer, then **`*ptr`** shall not be a null pointer. The value of **`*s1max`** shall not be greater than **`RSIZE_MAX`**. The end of the token found shall occur within the first **`*s1max`** wide characters of **`s1`** for the first call, and shall occur within the first **`*s1max`** wide characters of where searching resumes on subsequent calls.

3 If there is a runtime-constraint violation, the **`wcstok_s`** function does not indirect through the **`s1`** or **`s2`** pointers, and does not store a value in the object pointed to by **`ptr`**.

Description

4 A sequence of calls to the **`wcstok_s`** function breaks the wide string pointed to by **`s1`** into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by **`s2`**. The fourth argument points to a caller-provided **`wchar_t`** pointer into which the **`wcstok_s`** function stores information necessary for it to continue scanning the same wide string.

5 The first call in a sequence has a non-null first argument and **`s1max`** points to an object whose value is the number of elements in the wide character array pointed to by the first argument. The first call stores an initial value in the object pointed to by **`ptr`** and updates the value pointed to by **`s1max`** to reflect the number of elements that remain in relation to **`ptr`**. Subsequent calls in the sequence have a null first argument and the objects pointed to by **`s1max`** and **`ptr`** are required to have the values stored by the previous call in the sequence, which are then updated. The separator wide string pointed to by **`s2`** may be different from call to call.

6 The first call in the sequence searches the wide string pointed to by **`s1`** for the first wide character that is *not* contained in the current separator wide string pointed to by **`s2`**. If no such wide character is found, then there are no tokens in the wide string pointed to by **`s1`** and the **`wcstok_s`** function returns a null pointer. If such a wide character is found, it is the start of the first token.

7 The **wcstok_s** function then searches from there for the first wide character in **s1** that *is* contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by **s1**, and subsequent searches in the same wide string for a token return a null pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token.

8 In all cases, the **wcstok_s** function stores sufficient information in the pointer pointed to by **ptr** so that subsequent calls, with a null pointer for **s1** and the unmodified pointer value for **ptr**, shall start searching just past the element overwritten by a null wide character (if any).

Returns

9 The **wcstok_s** function returns a pointer to the first wide character of a token, or a null pointer if there is no token or there is a runtime-constraint violation.

10 EXAMPLE

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
static wchar_t str1[] = L"?a???b,,,#c";
static wchar_t str2[] = L"\t \t";
wchar_t *t, *ptr1, *ptr2;
rsize_t max1 = wcslen(str1)+1;
rsize_t max2 = wcslen(str2)+1;

t = wcstok_s(str1, &max1, "?", &ptr1);     // t points to the token "a"
t = wcstok_s(NULL, &max1, ",", &ptr1);     // t points to the token "??b"
t = wcstok_s(str2, &max2, " \t", &ptr2);   // t is a null pointer
t = wcstok_s(NULL, &max1, "#,", &ptr1);    // t points to the token "c"
t = wcstok_s(NULL, &max1, "?", &ptr1);     // t is a null pointer
```

K.3.9.2.4 Miscellaneous functions

K.3.9.2.4.1 The wcsnlen_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
size_t wcsnlen_s(const wchar_t *s, size_t maxsize);
```

Description

2 The **wcsnlen_s** function computes the length of the wide string pointed to by **s**.

Returns

3 If **s** is a null pointer,[452] then the **wcsnlen_s** function returns zero.

4 Otherwise, the **wcsnlen_s** function returns the number of wide characters that precede the terminating null wide character. If there is no null wide character in the first **maxsize** wide characters of **s** then **wcsnlen_s** returns **maxsize**. At most the first

maxsize wide characters of **s** shall be accessed by **wcsnlen_s**.

K.3.9.3 Extended multibyte/wide character conversion utilities

K.3.9.3.1 Restartable multibyte/wide character conversion functions

1 Unlike **wcrtomb**, **wcrtomb_s** does not permit the **ps** parameter (the pointer to the conversion state) to be a null pointer.

K.3.9.3.1.1 The wcrtomb_s function

Synopsis

2
```
#include <wchar.h>
errno_t wcrtomb_s(size_t * restrict retval,
     char * restrict s, rsize_t smax,
     wchar_t wc, mbstate_t * restrict ps);
```

Runtime-constraints

3 Neither **retval** nor **ps** shall be a null pointer. If **s** is not a null pointer, then **smax** shall not equal zero and shall not be greater than **RSIZE_MAX**. If **s** is not a null pointer, then **smax** shall be not be less than the number of bytes to be stored in the array pointed to by **s**. If **s** is a null pointer, then **smax** shall equal zero.

4 If there is a runtime-constraint violation, then **wcrtomb_s** does the following. If **s** is not a null pointer and **smax** is greater than zero and not greater than **RSIZE_MAX**, then **wcrtomb_s** sets **s[0]** to the null character. If **retval** is not a null pointer, then **wcrtomb_s** sets ***retval** to **(size_t)(-1)**.

Description

5 If **s** is a null pointer, the **wcrtomb_s** function is equivalent to the call

```
wcrtomb_s(&retval, buf, sizeof buf, L'\0', ps)
```

where **retval** and **buf** are internal variables of the appropriate types, and the size of **buf** is greater than **MB_CUR_MAX**.

6 If **s** is not a null pointer, the **wcrtomb_s** function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by **wc** (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by **s**. At most **MB_CUR_MAX** bytes are stored. If **wc** is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

---

452) Note that the **wcsnlen_s** function has no runtime-constraints. This lack of runtime-constraints along with the values returned for a null pointer or an unterminated wide string argument make **wcsnlen_s** useful in algorithms that gracefully handle such exceptional data.

7 If **wc** does not correspond to a valid multibyte character, an encoding error occurs: the **wcrtomb_s** function stores the value **(size_t)(-1)** into ***retval** and the conversion state is unspecified. Otherwise, the **wcrtomb_s** function stores into ***retval** the number of bytes (including any shift sequences) stored in the array pointed to by **s**.

Returns

8 The **wcrtomb_s** function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

K.3.9.3.2 Restartable multibyte/wide string conversion functions

1 Unlike **mbsrtowcs** and **wcsrtombs**, **mbsrtowcs_s** and **wcsrtombs_s** do not permit the **ps** parameter (the pointer to the conversion state) to be a null pointer.

K.3.9.3.2.1 The mbsrtowcs_s function

Synopsis

2
```
#include <wchar.h>
errno_t mbsrtowcs_s(size_t * restrict retval,
     wchar_t * restrict dst, rsize_t dstmax,
     const char ** restrict src, rsize_t len,
     mbstate_t * restrict ps);
```

Runtime-constraints

3 None of **retval**, **src**, ***src**, or **ps** shall be null pointers. If **dst** is not a null pointer, then neither **len** nor **dstmax** shall be greater than **RSIZE_MAX**. If **dst** is a null pointer, then **dstmax** shall equal zero. If **dst** is not a null pointer, then **dstmax** shall not equal zero. If **dst** is not a null pointer and **len** is not less than **dstmax**, then a null character shall occur within the first **dstmax** multibyte characters of the array pointed to by ***src**.

4 If there is a runtime-constraint violation, then **mbsrtowcs_s** does the following. If **retval** is not a null pointer, then **mbsrtowcs_s** sets ***retval** to **(size_t)(-1)**. If **dst** is not a null pointer and **dstmax** is greater than zero and less than **RSIZE_MAX**, then **mbsrtowcs_s** sets **dst[0]** to the null wide character.

Description

5 The **mbsrtowcs_s** function converts a sequence of multibyte characters that begins in the conversion state described by the object pointed to by **ps**, from the array indirectly pointed to by **src** into a sequence of corresponding wide characters. If **dst** is not a null pointer, the converted characters are stored into the array pointed to by **dst**. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if **dst** is not a null pointer) when **len** wide

characters have been stored into the array pointed to by **dst**.[453)] If **dst** is not a null pointer and no null wide character was stored into the array pointed to by **dst**, then **dst[len]** is set to the null wide character. Each conversion takes place as if by a call to the **mbrtowc** function.

6 If **dst** is not a null pointer, the pointer object pointed to by **src** is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multibyte character converted (if any). If conversion stopped due to reaching a terminating null character and if **dst** is not a null pointer, the resulting state described is the initial conversion state.

7 Regardless of whether **dst** is or is not a null pointer, if the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the **mbsrtowcs_s** function stores the value **(size_t)(-1)** into ***retval** and the conversion state is unspecified. Otherwise, the **mbsrtowcs_s** function stores into ***retval** the number of multibyte characters successfully converted, not including the terminating null character (if any).

8 All elements following the terminating null wide character (if any) written by **mbsrtowcs_s** in the array of **dstmax** wide characters pointed to by **dst** take unspecified values when **mbsrtowcs_s** returns.[454)]

9 If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

10 The **mbsrtowcs_s** function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

K.3.9.3.2.2 The wcsrtombs_s function

Synopsis

11
```
#include <wchar.h>
errno_t wcsrtombs_s(size_t * restrict retval,
     char * restrict dst, rsize_t dstmax,
     const wchar_t ** restrict src, rsize_t len,
     mbstate_t * restrict ps);
```

453) Thus, the value of **len** is ignored if **dst** is a null pointer.

454) This allows an implementation to attempt converting the multibyte string before discovering a terminating null character did not occur where required.

Runtime-constraints

12 None of **retval**, **src**, ***src**, or **ps** shall be null pointers. If **dst** is not a null pointer, then neither **len** nor **dstmax** shall be greater than **RSIZE_MAX**. If **dst** is a null pointer, then **dstmax** shall equal zero. If **dst** is not a null pointer, then **dstmax** shall not equal zero. If **dst** is not a null pointer and **len** is not less than **dstmax**, then the conversion shall have been stopped (see below) because a terminating null wide character was reached or because an encoding error occurred.

13 If there is a runtime-constraint violation, then **wcsrtombs_s** does the following. If **retval** is not a null pointer, then **wcsrtombs_s** sets ***retval** to **(size_t)(-1)**. If **dst** is not a null pointer and **dstmax** is greater than zero and less than **RSIZE_MAX**, then **wcsrtombs_s** sets **dst[0]** to the null character.

Description

14 The **wcsrtombs_s** function converts a sequence of wide characters from the array indirectly pointed to by **src** into a sequence of corresponding multibyte characters that begins in the conversion state described by the object pointed to by **ps**. If **dst** is not a null pointer, the converted characters are then stored into the array pointed to by **dst**. Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier in two cases:

— when a wide character is reached that does not correspond to a valid multibyte character;

— (if **dst** is not a null pointer) when the next multibyte character would exceed the limit of *n* total bytes to be stored into the array pointed to by **dst**. If the wide character being converted is the null wide character, then *n* is the lesser of **len** or **dstmax**. Otherwise, *n* is the lesser of **len** or **dstmax-1**.

If the conversion stops without converting a null wide character and **dst** is not a null pointer, then a null character is stored into the array pointed to by **dst** immediately following any multibyte characters already stored. Each conversion takes place as if by a call to the **wcrtomb** function.[455)]

15 If **dst** is not a null pointer, the pointer object pointed to by **src** is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

---

455) If conversion stops because a terminating null wide character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte. However, if the conversion stops before a terminating null wide character has been reached, the result will be null terminated, but might not end in the initial shift state.

16 Regardless of whether **dst** is or is not a null pointer, if the input conversion encounters a wide character that does not correspond to a valid multibyte character, an encoding error occurs: the **wcsrtombs_s** function stores the value **(size_t)(-1)** into ***retval** and the conversion state is unspecified. Otherwise, the **wcsrtombs_s** function stores into ***retval** the number of bytes in the resulting multibyte character sequence, not including the terminating null character (if any).

17 All elements following the terminating null character (if any) written by **wcsrtombs_s** in the array of **dstmax** elements pointed to by **dst** take unspecified values when **wcsrtombs_s** returns.[456)]

18 If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

19 The **wcsrtombs_s** function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

456) When **len** is not less than **dstmax**, the implementation might fill the array before discovering a runtime-constraint violation.

Annex L
(normative)
Analyzability

L.1 Scope

1 This annex specifies optional behavior that can aid in the analyzability of C programs.

2 An implementation that defines **`__STDC_ANALYZABLE__`** shall conform to the specifications in this annex.[457)]

L.2 Definitions

L.2.1

1 **out-of-bounds store**
an (attempted) access (3.1) that, at run time, for a given computational state, would modify (or, for an object declared **`volatile`**, fetch) one or more bytes that lie outside the bounds permitted by this Standard.

L.2.2

1 **bounded undefined behavior**
undefined behavior (3.4.3) that does not perform an out-of-bounds store.

2 NOTE 1 The behavior might perform a trap.

3 NOTE 2 Any values produced or stored might be indeterminate values.

L.2.3

1 **critical undefined behavior**
undefined behavior that is not bounded undefined behavior.

2 NOTE The behavior might perform an out-of-bounds store or perform a trap.

457) Implementations that do not define **`__STDC_ANALYZABLE__`** are not required to conform to these specifications.

L.3 Requirements

1 If the program performs a trap (3.19.5), the implementation is permitted to invoke a runtime-constraint handler. Any such semantics are implementation-defined.

2 All undefined behavior shall be limited to bounded undefined behavior, except for the following which are permitted to result in critical undefined behavior:

— An object is referred to outside of its lifetime (6.2.4).

— A store is performed to an object that has two incompatible declarations (6.2.7),

— A pointer is used to call a function whose type is not compatible with the referenced type (6.2.7, 6.3.2.3, 6.5.2.2).

— An lvalue does not designate an object when evaluated (6.3.2.1).

— The program attempts to modify a string literal (6.4.5).

— The operand of the unary **`*`** operator has an invalid value (6.5.3.2).

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary **`*`** operator that is evaluated (6.5.6).

— An attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type (6.7.3).

— An argument to a function or macro defined in the standard library has an invalid value or a type not expected by a function with variable number of arguments (7.1.4).

— The **`longjmp`** function is called with a **`jmp_buf`** argument where the most recent invocation of the **`setjmp`** macro in the same invocation of the program with the corresponding **`jmp_buf`** argument is nonexistent, or the invocation was from another thread of execution, or the function containing the invocation has terminated execution in the interim, or the invocation was within the scope of an identifier with variably modified type and execution has left that scope in the interim (7.13.2.1).

— The value of a pointer that refers to space deallocated by a call to the free or realloc function is used (7.22.3).

— A string or wide string utility function accesses an array beyond the end of an object (7.24.1, 7.29.4).

Bibliography

1. ‘‘The C Reference Manual’’ by Dennis M. Ritchie, a version of which was published in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., (1978). Copyright owned by AT&T.
2. *1984 /usr/group Standard* by the /usr/group Standards Committee, Santa Clara, California, USA, November 1984.
3. ANSI X3/TR−1−82 (1982), *American National Dictionary for Information Processing Systems*, Information Processing Systems Technical Report.
4. ANSI/IEEE 754−1985, *American National Standard for Binary Floating-Point Arithmetic*.
5. ANSI/IEEE 854−1988, *American National Standard for Radix-Independent Floating-Point Arithmetic*.
6. IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems, second edition* (previously designated IEC 559:1989).
7. ISO 31−11:1992, *Quantities and units — Part 11: Mathematical signs and symbols for use in the physical sciences and technology*.
8. ISO/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange*.
9. ISO/IEC 2382−1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*.
10. ISO 4217:1995, *Codes for the representation of currencies and funds*.
11. ISO 8601:1988, *Data elements and interchange formats — Information interchange — Representation of dates and times*.
12. ISO/IEC 9899:1990, *Programming languages — C*.
13. ISO/IEC 9899/COR1:1994, *Technical Corrigendum 1*.
14. ISO/IEC 9899/COR2:1996, *Technical Corrigendum 2*.
15. ISO/IEC 9899/AMD1:1995, *Amendment 1 to ISO/IEC 9899:1990 C Integrity*.
16. ISO/IEC 9899:1999, *Programming languages — C*.
17. ISO/IEC 9899:1999/Cor.1:2001, *Technical Corrigendum 1*.
18. ISO/IEC 9899:1999/Cor.2:2004, *Technical Corrigendum 2*.
19. ISO/IEC 9899:1999/Cor.3:2007, *Technical Corrigendum 3*.

20. ISO/IEC 9945−2:1993, *Information technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities*.
21. ISO/IEC TR 10176:1998, *Information technology — Guidelines for the preparation of programming language standards*.
22. ISO/IEC 10646−1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*.
23. ISO/IEC 10646−1/COR1:1996, *Technical Corrigendum 1 to ISO/IEC 10646−1:1993*.
24. ISO/IEC 10646−1/COR2:1998, *Technical Corrigendum 2 to ISO/IEC 10646−1:1993*.
25. ISO/IEC 10646−1/AMD1:1996, *Amendment 1 to ISO/IEC 10646−1:1993 Transformation Format for 16 planes of group 00 (UTF−16)*.
26. ISO/IEC 10646−1/AMD2:1996, *Amendment 2 to ISO/IEC 10646−1:1993 UCS Transformation Format 8 (UTF−8)*.
27. ISO/IEC 10646−1/AMD3:1996, *Amendment 3 to ISO/IEC 10646−1:1993*.
28. ISO/IEC 10646−1/AMD4:1996, *Amendment 4 to ISO/IEC 10646−1:1993*.
29. ISO/IEC 10646−1/AMD5:1998, *Amendment 5 to ISO/IEC 10646−1:1993 Hangul syllables*.
30. ISO/IEC 10646−1/AMD6:1997, *Amendment 6 to ISO/IEC 10646−1:1993 Tibetan*.
31. ISO/IEC 10646−1/AMD7:1997, *Amendment 7 to ISO/IEC 10646−1:1993 33 additional characters*.
32. ISO/IEC 10646−1/AMD8:1997, *Amendment 8 to ISO/IEC 10646−1:1993*.
33. ISO/IEC 10646−1/AMD9:1997, *Amendment 9 to ISO/IEC 10646−1:1993 Identifiers for characters*.
34. ISO/IEC 10646−1/AMD10:1998, *Amendment 10 to ISO/IEC 10646−1:1993 Ethiopic*.
35. ISO/IEC 10646−1/AMD11:1998, *Amendment 11 to ISO/IEC 10646−1:1993 Unified Canadian Aboriginal Syllabics*.
36. ISO/IEC 10646−1/AMD12:1998, *Amendment 12 to ISO/IEC 10646−1:1993 Cherokee*.
37. ISO/IEC 10967−1:1994, *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*.

38. ISO/IEC TR 19769:2004, *Information technology — Programming languages, their environments and system software interfaces — Extensions for the programming language C to support new character data types*.

39. ISO/IEC TR 24731−1:2007, *Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library — Part 1: Bounds-checking interfaces*.

Index

 *