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.

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 3. Accordingly, annexes F and I form a normative part of this standard; this foreword, the introduction, notes, footnotes, examples, annexes A, B, C, D, E, G, H, J, K, the bibliography, and the index are for information only.

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 International Standard ISO/IEC 9899 was prepared by Joint Technical Committee ISO/IEC JTC 1, ‘‘Information Technology’’, subcommittee 22, ‘‘Programming languages, their environments and system software interfaces’’.

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 [7.26]) is discouraged.

3 This International Standard is divided into four major subdivisions:

— the introduction and preliminary elements;

— the characteristics of environments that translate and execute C programs;

— the language syntax, constraints, and semantics;

— the library facilities.

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*.

Contents

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 normative documents contain provisions which, through reference in this text, constitute provisions of this International Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

2 ISO/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange*.

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

4 ISO 4217:1995, *Codes for the representation of currencies and funds*.

5 ISO 8601:1988, *Data elements and interchange formats — Information interchange — Representation of dates and times*.

6 ISO/IEC 10646:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*.

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

3. Terms and definitions

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.

3.1

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

3.2

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

3.3

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.

3.4

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*.

3.5

1 **character**
bit representation that fits in a byte *

3.6

1 **constraints**
restrictions, both syntactic and semantic, by which the exposition of language elements is to be interpreted

3.7

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

3.8

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

3.9

1 **forward references**
references to later subclauses of this International Standard that contain additional information relevant to this subclause

3.10

1 **implementation**
a 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

3.11

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.

3.12

1 **implementation limits**

restrictions imposed upon programs by the implementation

3.13

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.

3.14

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.

3.15

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**

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

3.18

1 **undefined behavior**

behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately valued objects, 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.

3.19

1 **unspecified behavior**

behavior where this International Standard provides two or more possibilities and imposes no 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.

Forward references: bitwise shift operators (6.5.7), expressions (6.5), function calls (6.5.2.2), the `islower` function (7.4.1.6), localization (7.11).

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 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.[2] 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 that does not use complex types and 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>**, **<stdarg.h>**, **<stdbool.h>**, **<stddef.h>**, and **<stdint.h>**. A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.[3)]

7 A *conforming program* is one that is acceptable to a conforming implementation.[4)]

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), characteristics of floating types **<float.h>** (7.7), alternative spellings **<iso646.h>** (7.9), sizes of integer types **<limits.h>** (7.10), variable arguments **<stdarg.h>** (7.15), boolean type and values **<stdbool.h>** (7.16), common definitions **<stddef.h>** (7.17), integer types **<stdint.h>** (7.18).

---

2) A strictly conforming program can use conditional features (such as those in annex F) provided the use is guarded by a **#ifdef** directive with the appropriate macro. For example:

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

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

4) 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: conditional inclusion (6.10.1), linkages of identifiers (6.2.2), source file inclusion (6.10.2), 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.[5)]

1. Physical source file multibyte characters are mapped 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.
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. If, as a result, a character sequence that matches the syntax of a universal character name is produced, the behavior is undefined. 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.

5) Implementations shall behave as if these separate phases occur, even though many are typically folded together in practice.

3. The source file is decomposed into preprocessing tokens[6] 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, escape sequence, and universal character name 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.
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).

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.[7]

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

6) 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.

7) 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.

```
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 in static storage 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: initialization (6.7.8).

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.

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;[8] or in some other implementation-defined manner.

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

8) 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.

— 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).

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;[9)] 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 (7.20.4.3).

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*,[10)] which are changes in the state of the execution environment. Evaluation of an expression may produce side effects. At certain specified points in the execution sequence called *sequence points*, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place. (A summary of the sequence points is given in annex C.)

---

9) In accordance with 6.2.4, objects with automatic storage duration declared in **main** will no longer have storage guaranteed to be reserved in the former case even where they would in the latter.

3 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 calling a function or accessing a volatile object).

4 When the processing of the abstract machine is interrupted by receipt of a signal, only the values of objects as of the previous sequence point may be relied on. Objects that may be modified between the previous sequence point and the next sequence point need not have received their correct values yet.

5 An instance of each object with automatic storage duration is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).

6 The least requirements on a conforming implementation are:

— At sequence points, volatile objects are stable in the sense that previous accesses are complete and subsequent accesses have not yet occurred.

— 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 7.19.3. 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.

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

10) 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.

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) expressions;
```

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.8).

```
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: compound statement, or block (6.8.2), expressions (6.5), files (7.19.3), sequence points (6.5, 6.8), the `signal` function (7.14), type qualifiers (6.7.3).

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, and the set interpreted in the execution environment. The values of the members of the execution character set are implementation-defined; any additional members beyond those required by this subclause are locale-specific.

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 at least 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 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 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 All occurrences in a source file of the following sequences of three characters (called *trigraph sequences*[11]) are 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 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 single-byte characters defined in 5.2.1 shall be present.

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

— 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.

— A byte with all bits zero shall not occur in the second or subsequent bytes of a 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.

11) 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.

— 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** or **fputwc** function would appear. The intent of writing a printable character (as defined by the **isprint** or **iswprint** 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 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. The active position shall not be changed.

\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 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 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 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 (7.4.1.7), the **fputc** function (7.19.7.3), the **fputwc** functions (7.24.3.3), the **iswprint** function (7.25.2.1.7).

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:[12)]

— 127 nesting levels of compound statements, iteration statements, and selection statements

— 63 nesting levels of conditional inclusion

— 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or incomplete 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 character short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a character 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)

— 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

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

— 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 character string literal or wide 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 A conforming implementation shall 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>**.

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 (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**.[13] The value **UCHAR_MAX+1** shall equal 2 raised to the power **CHAR_BIT**.

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.[14] The following parameters are used to

13) See 6.2.5.

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 normalized floating-point number x ($f_1 > 0$ if $x \neq 0$) is defined by the following model:

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

3 Floating types may include values that are not normalized floating-point numbers, for example subnormal floating-point numbers ($x \neq 0$, $e = e_{\min}$, $f_1 = 0$), infinities, and NaNs.[15] A *NaN* is an encoding signifying Not-a-Number. A *quiet NaN* propagates through almost every arithmetic operation without raising an exception; a *signaling NaN* generally raises an exception when occurring as an arithmetic operand.[16]

4 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. The implementation may state that the accuracy is unknown.

5 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**.

6 The rounding mode for floating-point addition is characterized by the value of **FLT_ROUNDS**:[17]

−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

14) 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.

15) Although they are stored in floating types, infinities and NaNs are not floating-point numbers.

16) 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.

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

behavior.

7 The values of operations 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 value of **FLT_EVAL_METHOD**:[18)]

- **−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.

8 The values given in the following list shall be replaced by implementation-defined constant expressions with 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**

— 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 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,

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

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,

18) 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**.

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

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, $\lceil \log_{10} b^{e_{\min}-1} \rceil$

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, $\lfloor \log_{10}((1 - b^{-p}) \times b^{e_{\max}}) \rfloor$

FLT_MAX_10_EXP	**+37**
DBL_MAX_10_EXP	**+37**
LDBL_MAX_10_EXP	**+37**

9 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:

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

FLT_MAX	**1E+37**
DBL_MAX	**1E+37**
LDBL_MAX	**1E+37**

10 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:

— 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
```

11 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**:

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

```
FLT_RADIX                        16
FLT_MANT_DIG                      6
FLT_EPSILON          9.53674316E-07F
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
```

12 EXAMPLE 2 The following describes floating-point representations that also meet the requirements for single-precision and double-precision normalized numbers in IEC 60559,[19] and the appropriate values in a **<float.h>** header for types **float** and **double**:

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

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

19) 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_RADIX                          2
DECIMAL_DIG                       17
FLT_MANT_DIG                      24
FLT_EPSILON           1.19209290E-07F // decimal constant
FLT_EPSILON                  0X1P-23F // hex constant
FLT_DIG                            6
FLT_MIN_EXP                     -125
FLT_MIN               1.17549435E-38F // decimal constant
FLT_MIN                     0X1P-126F // hex constant
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_DIG                           15
DBL_MIN_EXP                    -1021
DBL_MIN    2.2250738585072014E-308 // decimal constant
DBL_MIN                     0X1P-1022 // hex constant
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), mathematics **<math.h>** (7.12), integer types **<stdint.h>** (7.18).

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 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). Label names shall be unique within a function.

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 **`}`** that closes 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 be a strict subset of 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.

Forward references: compound statement, or block (6.8.2), declarations (6.7), enumeration specifiers (6.7.2.2), function calls (6.5.2.2), function declarators (including prototypes) (6.7.5.3), function definitions (6.9.1), the **goto** statement (6.8.6.1), labeled statements (6.8.1), name spaces of identifiers (6.2.3), scope of macro definitions (6.10.3.5), source file inclusion (6.10.2), tags (6.7.2.3), type specifiers (6.7.2).

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*. 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.[20]

4 For an identifier declared with the storage-class specifier **extern** in a scope in which a prior declaration of that identifier is visible,[21] 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,

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

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

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: compound statement, or block (6.8.2), declarations (6.7), expressions (6.5), external definitions (6.9).

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[22)] 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).

6.2.4 Storage durations of objects

1 An object has a *storage duration* that determines its lifetime. There are three storage durations: static, automatic, and allocated. Allocated storage is described in 7.20.3.

2 An object whose identifier is declared with external or internal linkage, or with the storage-class specifier **`static`** has *static storage duration*. For such an object, storage is reserved and its stored value is initialized only once, prior to program startup. The object exists, has a constant address, and retains its last-stored value throughout the execution of the entire program.[23)]

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

23) 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.

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

3 An object whose identifier is declared with no linkage and without the storage-class specifier **`static`** has *automatic storage duration*. For objects that do not have a variable length array type, storage is guaranteed to be reserved for a new instance of the object on each entry into the block with which it is associated; the initial value of the object is indeterminate. If an initialization is specified for the object, it is performed each time the declaration is reached in the execution of the block; otherwise, the value becomes indeterminate each time the declaration is reached. Storage for the object is no longer guaranteed to be reserved when execution of the block ends in any way. (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.)

4 For objects that do have a variable length array type, storage is guaranteed to be reserved for a new instance of the object each time the declaration is reached in the execution of the program. The initial value of the object is indeterminate. Storage for the object is no longer guaranteed to be reserved when the execution of the program leaves the scope of the declaration.[24)]

5 If an object is referred to when storage is not reserved for it, the behavior is undefined. The value of a pointer that referred to an object whose storage is no longer reserved is indeterminate. During the time that its storage is reserved, an object has a constant address.

Forward references: compound statement, or block (6.8.2), function calls (6.5.2.2), declarators (6.7.5), array declarators (6.7.5.2), initialization (6.7.8).

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), *function types* (types that describe functions), and *incomplete types* (types that describe objects but lack information needed to determine their sizes).

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 required source character set enumerated in 5.2.1 is stored in a **`char`** object, its value is guaranteed to be positive. 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*.[25)] The standard and extended

24) 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.

signed integer types are collectively called *signed integer types*.[26)]

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*.

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 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.[27)] 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 unsigned integer type.

10 There are three *real floating types*, designated as **`float`**, **`double`**, and **`long double`**. 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`**.

11 There are three *complex types*, designated as **`float _Complex`**, **`double _Complex`**, and **`long double _Complex`**.[28)] 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

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

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

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

28) A specification for imaginary types is in informative annex G.

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*. Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.[29)]

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`**.[30)]

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

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

— An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*.[31)] 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.

29) 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.

30) **`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.

31) Since object types do not include incomplete types, an array of incomplete type cannot be constructed.

— 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, an object type, or an incomplete 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’’.

20 These methods of constructing derived types can be applied recursively.

21 Integer and floating types are collectively called *arithmetic types*. Arithmetic types and pointer types are collectively called *scalar types*. Array and structure types are collectively called *aggregate types*.[32)]*

22 Each arithmetic type belongs to one *type*domain. The *real type domain* comprises the real types. The *complex type domain* comprises the complex types.

23 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 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. A structure type containing a flexible array member is an incomplete type that cannot be completed.

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,[33)] 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.[27)] A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

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

33) See 6.7.3 regarding qualified array and function types.

27 A pointer to **void** shall have the same representation and alignment requirements as a pointer to a character type. Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements.[27)] 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.

28 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-qualified **float**" and is a pointer to a qualified type.

29 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: character constants (6.4.4.4), compatible type and composite type (6.2.7), declarations (6.7), tags (6.7.2.3), type qualifiers (6.7.3). *

6.2.6 Representations of types

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

6.2.6.1 General

1 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.

2 Values stored in objects of type **unsigned char** shall be represented using a pure binary notation.[34)]

3 Values stored in 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. Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations.

4 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 accessed by an lvalue expression that does not have character type, the behavior is undefined. If such a representation is

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

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.[35)] Such a representation is called a *trap representation*.

5 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.[36)] The values of padding bytes shall not affect whether the value of such an object is a trap representation. Those bits of a structure or union object that are in the same byte as a bit-field member, but are not part of that member, shall similarly not affect whether the value of such an object is a trap representation.

6 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, but the value of the union object shall not thereby become a trap representation.

7 Where an operator is applied to a value which has more than one object representation, which object representation is used shall not affect the value of the result. 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.

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.[37)]

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; there shall be exactly one sign bit. Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type (if there are M value bits in the signed type and N in the unsigned type, then $M \leq N$). If the sign bit is zero, it shall not affect the resulting value. If the sign bit is one, then the value shall be modified in one of the following ways:

---

35) 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.

36) Thus, for example, structure assignment may be implemented element-at-a-time or via **`memcpy`**.

37) 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 exception 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 corresponding value with sign bit 0 is negated;

— the sign bit has the value -2^N;

— the sign bit has the value $1 - 2^N$.

3 The values of any padding bits are unspecified.[37] 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.

4 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.

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 6.7.5 for declarators.[38] 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 types, 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, and such that if one member of a corresponding pair is declared with a name, the other member 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 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, the composite type is that type.

— 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.

— 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.

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

4 For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible,[39] if the prior declaration specifies internal or external linkage, the type of the identifier at the later declaration becomes the composite type.

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]);
```

Forward references: declarators (6.7.5), enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1), type definitions (6.7.7), type qualifiers (6.7.3), type specifiers (6.7.2).

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

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.

— 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 whose integer conversion rank is less than 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, the value is converted to an **int**; otherwise, it is converted to an **unsigned int**. These are called the *integer promotions*.[40] 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.

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.

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

6.3.1.4 Real floating and integer

1 When a finite value of real floating type is converted to 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.[41]

2 When a value of integer type is converted to real floating type, 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 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.

40) 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.

41) 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).

6.3.1.5 Real floating types

1 When a **`float`** is promoted to **`double`** or **`long double`**, or a **`double`** is promoted to **`long double`**, its value is unchanged.

2 When a **`double`** is demoted to **`float`** or a **`long double`** to **`double`** or **`float`**, if the value being converted is outside the range of values that can be represented, the behavior is undefined. 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 value, chosen in an implementation-defined manner.

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 determined by the operator. 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`**.[42)]

---

42) 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).

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 precision and range than that required by the type; the types are not changed thereby.[43)]

6.3.2 Other operands

6.3.2.1 Lvalues and function designators

1 An *lvalue* is an expression with an object type or an incomplete type other than **`void`**;[44)] 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 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

---

43) The cast and assignment operators are still required to perform their specified conversions as described in 6.3.1.4 and 6.3.1.5.

44) 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.

object (and is no longer an lvalue). If the lvalue has qualified type, the value has the unqualified 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.

3 Except when it is the operand of the **`sizeof`** 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[45)] or the unary **`&`** operator, a function designator with type ''function returning *type*'' is converted to an expression that 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>`** (7.17), initialization (6.7.8), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), the **`sizeof`** operator (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 incomplete or object type. A pointer to any incomplete or 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*.[46)] If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type. Such a pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.

---

45) Because this conversion does not occur, the operand of the **`sizeof`** operator remains a function designator and violates the constraint in 6.5.3.4.

46) The macro **`NULL`** is defined in **`<stddef.h>`** as a null pointer constant; see 7.17.

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. The result is implementation-defined, might not be properly aligned, and might not point to an entity of the referenced type.47)

6 Any pointer type may be converted to an integer type; 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.

7 A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. If the resulting pointer is not correctly aligned48) for the pointed-to 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 pointed-to type, the behavior is undefined.

Forward references: cast operators (6.5.4), equality operators (6.5.9), simple assignment (6.5.16.1).

47) 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.

48) 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 *universal-character-name* that cannot be one of the above
- 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 token 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.[49] 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.

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: a header name preprocessing token is only recognized within a **`#include`** preprocessing directive, and within such a

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

directive, 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`	**`enum`**	**`restrict`**	**`unsigned`**
`break`	**`extern`**	**`return`**	**`void`**
`case`	**`float`**	**`short`**	**`volatile`**
`char`	**`for`**	**`signed`**	**`while`**
`const`	**`goto`**	**`sizeof`**	**`_Bool`**
`continue`	**`if`**	**`static`**	**`_Complex`**
`default`	**`inline`**	**`struct`**	**`_Imaginary`**
`do`	**`int`**	**`switch`**	
`double`	**`long`**	**`typedef`**	
`else`	**`register`**	**`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.

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. Lower-case and upper-case 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 annex I.[50)] The initial nondigit character shall not be a universal character name designating a digit. An implementation may allow multibyte characters that are not part of the required source character set to appear in identifiers; which characters and their correspondence to universal character names is implementation defined.

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.

50) 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.

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.[51]

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).

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 short identifier in the range 00000000 through 00000020, 0000007F through 0000009F, or 0000D800 through 0000DFFF inclusive. A universal character name shall not designate a character in the required character set.

51) Note that 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.

Description

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

Semantics

4 The universal character name **`\U`***nnnnnnnn* designates the character whose character short identifier (as specified by ISO/IEC 10646) is *nnnnnnnn*. Similarly, the universal character name **`\u`***nnnn* designates the character whose character short identifier is 0000*nnnn*.

6.4.4 Constants

Syntax

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

Constraints

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

Semantics

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

6.4.4.1 Integer constants

Syntax

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

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.

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`**.

Recommended practice

5 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.

6 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.[52)]

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* **`'`**

---

52) The specification for the library functions recommends more accurate conversion than required for floating constants (see 7.20.1.3).

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'`** or **`'ab'`**. A wide character constant is the same, except prefixed by the letter **`L`**. 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, graphic characters not in the required 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`**.[53)]

Constraints

9 The value of an octal or hexadecimal escape sequence shall be in the range of representable values for the type **`unsigned char`** for an integer character constant, or the unsigned type corresponding to **`wchar_t`** for a wide character constant.

Semantics

10 An integer character constant has type **`int`**. The value of an integer character constant containing a single character that maps to a member of the basic execution character set 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, or containing a character or escape sequence not represented in the basic execution character set, 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 has type **`wchar_t`**, an integer type defined in the **`<stddef.h>`** header. The value of a wide character constant containing a single multibyte character that maps to a member of the extended execution character set is the *wide character* (code) corresponding to that multibyte character, as defined by the **`mbtowc`** function, with an implementation-defined current locale. The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined.

---

53) 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" (6.11.1).

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 .

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>`** (7.17), the **`mbtowc`** function (7.20.7.2). *

6.4.5 String literals

Syntax

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

s-char-sequence:
  s-char
  s-char-sequence s-char

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

Description

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

3 The same considerations apply to each element of the sequence in a character string literal or a wide string literal as if it were in an integer character constant or a wide character constant, 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

4 In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character and wide string literal tokens are concatenated into a single multibyte character sequence. If any of the tokens are wide string literal tokens, the resulting multibyte character sequence is treated as a wide string literal; otherwise, it is treated as a character string literal.

5 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.[54] 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 wide string literals, 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. The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined.

6 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.

7 EXAMPLE 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.

Forward references: common definitions **<stddef.h>** (7.17).

6.4.6 Punctuators

Syntax

1 *punctuator:* one of

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

54) A character 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.

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, these six tokens

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

behave, respectively, the same as these six tokens

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

except for their spelling.[55]

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.

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

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 sequence between the **"** delimiters, the behavior is undefined.[56] A header name preprocessing token is recognized only within a **#include** preprocessing directive.

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 letters, underscores, digits, periods, and **e+**, **e-**, **E+**, **E-**, **p+**, **p-**, **P+**, or **P-** character sequences.

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.

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

6.4.9 Comments

1 Except within a character constant, a string literal, or a comment, the characters **/*** introduce a comment. The contents of a comment are examined only to identify multibyte characters and to find the characters ***/** that terminate it.[57)]

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 1

```
"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;
```

57) 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.

2 Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored.[58] Informative annex D presents an algorithm for determining whether an expression or set of expressions meets these requirements.

3 The grouping of operators and operands is indicated by the syntax.[59] Except as specified later (for the function-call **()**, **&&**, **||**, **?:**, and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified.

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 return values that depend on the internal representations of integers, and have implementation-defined and undefined aspects for signed types.

5 If an *exception* 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.

6 The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[60] 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

---

58) This paragraph renders undefined statement expressions such as

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

while allowing

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

59) 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.

60) Allocated objects have no declared type.

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:[61)]

— 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 an atomic operation, thereby omitting rounding errors implied by the source code and the expression evaluation method.[62)] The **FP_CONTRACT** pragma in **<math.h>** provides a way to disallow contracted expressions. Otherwise, whether and how expressions are contracted is implementation-defined.[63)]

6.5.1 Primary expressions

Syntax

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

Semantics

---

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

62) A contracted expression might also omit the raising of floating-point exception flags.

63) 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.

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).[64)]

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.

Forward references: declarations (6.7).

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 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).

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

3 Successive subscript operators designate an element of a multidimensional array object. If **E** is an n-dimensional array ($n \geq 2$) with dimensions $i \times j \times \ldots \times k$, then **E** (used as other than an lvalue) is converted to a pointer to an $(n-1)$-dimensional array with dimensions $j \times \ldots \times k$. If the unary `*` operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the pointed-to $(n-1)$-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 (6.7.5.2).

6.5.2.2 Function calls

Constraints

1 The expression that denotes the called function[65] shall have type pointer to function returning **void** or returning an 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 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.[66]

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

66) 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 converted to a parameter with a pointer type as described in 6.9.1.

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**. If an attempt is made to modify the result of a function call or to access it after the next sequence point, the behavior is undefined.

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 agree with 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;

— one type is pointer to void and the other is a pointer to a character type.

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 The order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified, but there is a sequence point before the actual call.

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) (6.7.5.3), 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 a 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 qualified or unqualified structure’’ or ‘‘pointer to qualified or unqualified union’’, and the second operand shall name a member of the type pointed to.

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, 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.[67] 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 With one exception, if the value of a member of a union object is used when the most recent store to the object was to a different member, the behavior is implementation-defined.[68] 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.

6 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.

7 EXAMPLE 2 In:

---

67) 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`**.

68) The ‘‘byte orders’’ for scalar types are invisible to isolated programs that do not indulge in type punning (for example, by assigning to one member of a union and inspecting the storage by accessing another member that is an appropriately sized array of character type), but have to be accounted for when conforming to externally imposed storage layouts.

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

the various members have the types:

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

8 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 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. After the result is obtained, the value of the operand 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 side effect of updating the stored value of the operand shall occur between the previous and the next sequence point.

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 an object type or an array of unknown size.

2 No initializer shall attempt to provide a value for an object not contained within the entire unnamed object specified by the compound literal.

3 If the compound literal occurs outside the body of a function, the initializer list shall consist of constant expressions.

Semantics

4 A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a compound literal. It provides an unnamed object whose value is given by the initializer list.[69)]

5 If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in 6.7.7, 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.

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

7 All the semantic rules and constraints for initializer lists in 6.7.8 are applicable to compound literals.[70)]

8 String literals, and compound literals with const-qualified types, need not designate distinct objects.[71)]

9 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.

10 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

---

69) 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.

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

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

unnamed object has automatic storage duration.

11 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});
```

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

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

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

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

13 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.

14 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.

15 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);
```

16 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;

      while (j < 2)
            q = p, p = &((struct s){ j++ });
      return p == q && q->i == 1;
}
```

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

17 Note that if a **for** loop were used instead of a **while** loop, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around **p** would be pointing to an object which is no longer guaranteed to exist, which would result in undefined behavior.

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* **)**

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 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 result of the unary **&** (address-of) operator is a pointer to the object or function designated by 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.

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.[72)]

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.

---

72) 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 automatic storage duration object when execution of the block with which the object is associated has terminated.

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)`.

Forward references: characteristics of floating types `<float.h>` (7.7), sizes of integer types `<limits.h>` (7.10).

6.5.3.4 The `sizeof` operator

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 lvalue that designates a bit-field object.

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 When 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.[73)] 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.

4 The value of the result is implementation-defined, and its type (an unsigned integer type) is `size_t`, defined in the `<stddef.h>` header.

5 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`.

---

73) When applied to a parameter declared to have array or function type, the `sizeof` operator yields the size of the pointer obtained by converting as in 6.3.2.1; see 6.9.1.

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

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

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

```
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>** (7.17), declarations (6.7), structure and union specifiers (6.7.2.1), type names (6.7.6), array declarators (6.7.5.2).

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 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.

Semantics

4 Preceding an expression by a parenthesized type name converts the value of the expression to the named type. This construction is called a *cast*.[74)] A cast that specifies no conversion has no effect on the type or value of an expression.[75)]

74) 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.

75) If the value of the expression is represented with greater precision or range 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.

Forward references: equality operators (6.5.9), function declarators (including prototypes) (6.7.5.3), simple assignment (6.5.16.1), type names (6.7.6).

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. If either operand has complex type, the result has complex type.

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.[76] If the quotient `a/b` is representable, the expression `(a/b)*b + a%b` shall equal `a`.

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 an object type and the other shall have integer type. (Incrementing is equivalent to adding 1.)

3 For subtraction, one of the following shall hold:

76) This is often called ‘‘truncation toward zero’’.

— both operands have arithmetic type;

— both operands are pointers to qualified or unqualified versions of compatible object types; or

— the left operand is a pointer to an 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. If either operand has complex type, the result has complex type.

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 a nonarray object 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.77)

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 (6.7.5.2), common definitions **<stddef.h>** (7.17).

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 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 $\mathbf{E1} \times 2^{\mathbf{E2}}$, reduced modulo one more than the maximum value representable in the result type. If **E1** has a signed type and nonnegative value, and $\mathbf{E1} \times 2^{\mathbf{E2}}$ is representable in the result type, then that is

---

77) 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.

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 **E1** divided by the quantity, 2 raised to the power **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;

— both operands are pointers to qualified or unqualified versions of compatible object types; or

— both operands are pointers to qualified or unqualified versions of compatible incomplete 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 a nonarray object 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 or incomplete 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.[78] 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 or incomplete 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.[79] 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 conversion to the complex type corresponding to the common real type determined by the usual arithmetic conversions are equal.

5 Otherwise, at least one operand is a pointer. If one operand is a null pointer constant, it is converted to the type of the other operand. If one operand is a pointer to an object or incomplete 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 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

78) 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`”.

79) Because of the precedences, `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth-value.

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.[80)]

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).

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).

80) 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, the effect of subsequent comparisons is also undefined.

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; there is a sequence point after the evaluation of the first operand. 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; there is a sequence point after the evaluation of the first operand. 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 compatible structure or union types;

— 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 or incomplete 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 after its evaluation. 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.[81] If an attempt is made to modify the result of a conditional operator or to access it after the next sequence point, the behavior is undefined.

5 If both the second and third operands have arithmetic type, the type that the usual arithmetic conversions would yield if 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 pointed-to by both operands. Furthermore, if both operands are pointers to compatible types or to differently qualified versions of compatible types, the result type is

81) A conditional expression does not yield an lvalue.

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**.

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, but is not an lvalue. The type of an assignment expression is the type of the left operand unless the left operand has qualified type, in which case it is the unqualified version of the type of the left operand. The side effect of updating the stored value of the left operand shall occur between the previous and the next sequence point.

4 The order of evaluation of the operands is unspecified. If an attempt is made to modify the result of an assignment operator or to access it after the next sequence point, the behavior is undefined.

6.5.16.1 Simple assignment

Constraints

1 One of the following shall hold:[82)]

— the left operand has qualified or unqualified arithmetic type and the right has arithmetic type;

— the left operand has a qualified or unqualified version of a structure or union type compatible with the type of the right;

— 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;

— one operand is a pointer to an object or incomplete 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; or

— the left operand is a pointer and the right is a null pointer constant.

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

4 EXAMPLE 1 In the program fragment

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

the **`int`** value returned by the function 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`**.

---

82) 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’’ which removes any type qualifiers from the type category of the expression.

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 a pointer to an object type and the right shall have integer type, or the left operand shall have qualified or unqualified arithmetic type and the right shall have arithmetic type.

2 For the other operators, 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** differs from the simple assignment expression **E1 = E1** *op* **(E2)** only in that the lvalue **E1** is evaluated only once.

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 after its evaluation. Then the right operand is evaluated; the result has its type and value.[83] If an attempt is made to modify the result of a comma operator or to

access it after the next sequence point, the behavior is undefined.

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 (6.7.8).

83) 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.[84)]

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 precision and range shall be at least as great as if the expression were being evaluated in the execution environment.

6 An *integer constant expression*[85)] shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, **`sizeof`** expressions whose results are integer constants, 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`** 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,

— a null pointer constant,

— an address constant, or

— an address constant for an object type plus or minus an integer constant expression.

---

84) The operand of a **`sizeof`** operator is usually not evaluated (6.5.3.4).

85) An integer constant expression is used to specify the size of a bit-field member of a structure, the value of an enumeration constant, the size of an array, or the value of a **`case`** constant. Further constraints that apply to the integer constant expressions used in conditional-inclusion preprocessing directives are discussed in 6.10.1.

8 An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, and **`sizeof`** expressions. Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to the **`sizeof`** operator.

9 An *address constant* is a null pointer, a pointer to an lvalue designating an object of static storage duration, or 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.[86)]

Forward references: array declarators (6.7.5.2), initialization (6.7.8).

86) 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$_{opt}$ **;**

declaration-specifiers:
  storage-class-specifier declaration-specifiers$_{opt}$
  type-specifier declaration-specifiers$_{opt}$
  type-qualifier declaration-specifiers$_{opt}$
  function-specifier declaration-specifiers$_{opt}$

init-declarator-list:
  init-declarator
  init-declarator-list **,** *init-declarator*

init-declarator:
  declarator
  declarator **=** *initializer*

Constraints

2 A 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 for tags 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;[87)]

— for an enumeration constant or typedef name, is the (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.

87) Function definitions have a different syntax, described in 6.9.1.

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.

Forward references: declarators (6.7.5), enumeration specifiers (6.7.2.2), initialization (6.7.8), tags (6.7.2.3).

6.7.1 Storage-class specifiers

Syntax

1 *storage-class-specifier:*
```
typedef
extern
static
auto
register
```

Constraints

2 At most, one storage-class specifier may be given in the declaration specifiers in a declaration.[88)]

Semantics

3 The **`typedef`** specifier is called a ‘‘storage-class specifier’’ for syntactic convenience only; it is discussed in 6.7.7. The meanings of the various linkages and storage durations were discussed in 6.2.2 and 6.2.4.

4 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.[89)]

5 The declaration of an identifier for a function that has block scope shall have no explicit storage-class specifier other than **`extern`**.

6 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.

---

88) See ‘‘future language directions’’ (6.11.2).

89) 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 operator that can be applied to an array declared with storage-class specifier **`register`** is **`sizeof`**.

Forward references: type definitions (6.7.7).

6.7.2 Type specifiers

Syntax

1 *type-specifier:*
  `void`
  `char`
  `short`
  `int`
  `long`
  `float`
  `double`
  `signed`
  `unsigned`
  `_Bool`
  `_Complex`
  `_Imaginary`
  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 sets (delimited by commas, when there is more than one set on a line); 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`**

— **`float _Imaginary`**

— **`double _Imaginary`**

— **`long double _Imaginary`**

— struct or union specifier

— enum specifier

— typedef name

3 The type specifiers **`_Complex`** and **`_Imaginary`** shall not be used if the implementation does not provide those types.[90)]

Semantics

4 Specifiers for structures, unions, and enumerations are discussed in 6.7.2.1 through 6.7.2.3. Declarations of typedef names are discussed in 6.7.7. The characteristics of the other types are discussed in 6.2.5.

5 Each of the comma-separated sets 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: enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1), tags (6.7.2.3), type definitions (6.7.7).

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

90) Implementations are not required to provide imaginary types. Freestanding implementations are not required to provide complex types.

struct-or-union:
  `struct`
  `union`

struct-declaration-list:
  struct-declaration
  struct-declaration-list struct-declaration

struct-declaration:
  specifier-qualifier-list struct-declarator-list **`;`**

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 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.

3 The expression that specifies the width of a bit-field shall be an integer constant expression that has nonnegative value that shall not exceed the number of bits in an object of the type that is specified if the colon and expression are omitted. If the value is zero, the declaration shall have no declarator.

Semantics

4 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.

5 Structure and union specifiers have the same form.

6 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 contains no named members, the behavior is undefined. The type is incomplete until after the **`}`** that terminates the list.

7 A member of a structure or union may have any object type other than a variably modified type.[91] 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*;[92] its width is preceded by a colon.

8 A bit-field shall have a type that is a qualified or unqualified version of **`_Bool`**, **`signed int`**, or **`unsigned int`**. A bit-field is interpreted as a signed or unsigned integer type consisting of the specified number of bits.[93] 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.

9 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.

10 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[94] As a special case, a bit-field structure member with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

11 Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

12 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.

13 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.

---

91) A structure or union can not contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3.

92) 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.

93) 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.

94) An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts.

14 There may be unnamed padding at the end of a structure or union.

15 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*, and the size of the structure shall be equal to the offset of the last element of an otherwise identical structure that replaces the flexible array member with an array of unspecified length.[95] When an lvalue whose type is a structure with a flexible array member is used to access an object, 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, then 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.

16 EXAMPLE Assuming that all array members are aligned the same, after the declarations:

```
struct s { int n; double d[]; };
struct ss { int n; double d[1]; };
```

the three expressions:

```
sizeof (struct s)
offsetof(struct s, d)
offsetof(struct ss, d)
```

have the same value. The structure **struct s** has a flexible array member **d**.

17 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 as if the identifiers had been declared as:

```
struct { int n; double d[8]; } *s1;
struct { int n; double d[5]; } *s2;
```

18 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:

95) The length is unspecified to allow for the fact that implementations may give array members different alignments according to their lengths.

```
double *dp;
dp = &(s1->d[0]);          // Permitted
*dp = 42;                  // Permitted
dp = &(s2->d[0]);          // Permitted
*dp = 42;                  // Undefined behavior
```

Forward references: 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.[96] 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 an integer type. The choice of type is | implementation-defined,[97] but shall be capable of representing the values of all the members of the enumeration. The enumerated type is incomplete until after the **}** that * terminates the list of enumerator declarations.

---

96) 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.

97) An implementation may delay the choice of which integer type until all enumeration constants have been seen.

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 A type specifier of the form

> **`enum`** *identifier*

without an enumerator list shall only appear after the type it specifies is completed.

Semantics

3 All declarations of structure, union, or enumerated types that have the same scope and use the same tag declare the same type. The type is incomplete[98)] until the closing brace of the list defining the content, and complete thereafter.

4 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.

5 A type specifier of the form

> *struct-or-union identifier*$_{opt}$ **`{`** *struct-declaration-list* **`}`**

or

> **`enum`** *identifier* **`{`** *enumerator-list* **`}`**

or

> **`enum`** *identifier* **`{`** *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,[99)] the type specifier also declares the identifier to be the tag of that type.

98) 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.

6 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.[100)]

7 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.[100)]

8 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.

9 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**.

10 The following alternative formulation uses the **typedef** mechanism:

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

99) 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.

100) A similar construction with **enum** does not exist.

11 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 (6.7.5), array declarators (6.7.5.2), type definitions (6.7.7).

6.7.3 Type qualifiers

Syntax

1 *type-qualifier:*
const
restrict
volatile

Constraints

2 Types other than pointer types derived from object or incomplete types shall not be restrict-qualified.

Semantics

3 The properties associated with qualified types are meaningful only for expressions that are lvalues.[101)]

4 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.

5 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.[102)]

---

101) 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.

102) 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).

6 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.[103] What constitutes an access to an object that has volatile-qualified type is implementation-defined.

7 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.[104] The intended use of the **restrict** qualifier (like the **register** storage class) is to promote optimization, and deleting all instances of the qualifier from a conforming program does not change its meaning (i.e., observable behavior).

8 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.[105]

9 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.

10 EXAMPLE 1 An object declared

```
extern const volatile int real_time_clock;
```

may be modifiable by hardware, but cannot be assigned to, incremented, or decremented.

11 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;
```

103) 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.

104) 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.

105) Both of these can occur through the use of **typedef**s.

```
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 *"
```

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.

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**.[106)] Note that "based" is defined only for expressions with pointer types.

4 During each execution of **B**, let **A** be the array object that is determined dynamically by all accesses through pointer expressions based on **P**. Then *all* accesses to values of **A** shall be through pointer expressions based on **P**. Furthermore, if **P** is assigned the value of a pointer expression **E** that is based on another restricted pointer 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 during which storage is guaranteed to be reserved for an instance of an object that is associated with **B** and that has automatic storage duration. An access to a value means either fetching it or modifying it; expressions that are not evaluated do not access values.

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 the value of one of **a**, **b**, or **c**, then it is never accessed using the value of either of the other two.

106) 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.

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); // ok
      f(50, d +  1, d); // undefined behavior
}
```

10 EXAMPLE 3 The function parameter declarations

```
void h(int n, int * const restrict p,
      int * const q, int * const r)
{
      int i;
      for (i = 0; i < n; i++)
            p[i] = q[i] + r[i];
}
```

show how **const** can be used in conjunction with **restrict**. The **const** qualifiers imply, without the need to examine the body of **h**, that **q** and **r** cannot become based on **p**. The fact that **p** is restrict-qualified therefore implies that an object accessed through **p** is never accessed through either of **q** or **r**. This is the precise assertion required to optimize the loop. Note that a call of the form **h(100, a, b, b)** has defined behavior, which would not be true if all three of **p**, **q**, and **r** were restrict-qualified.

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; // ok
                int * restrict q2 = q1; // ok
                p1 = q2; // undefined behavior
                p2 = q2; // undefined behavior
        }
}
```

The 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`

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 storage duration, and shall not contain a reference to an identifier with internal linkage.

4 The **`inline`** function specifier shall not appear in a declaration of **`main`**.

Semantics

5 A function declared with an **`inline`** function specifier is an *inline function*. (The function specifier may appear more than once; the behavior is the same as if it appeared only once.) Making a function an inline function suggests that calls to the function be as fast as possible.[107)] The extent to which such suggestions are effective is implementation-defined.[108)]

6 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.[109)]

7 EXAMPLE 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

double convert(int is_fahr, double temp)
{
      /* A translator may perform inline substitutions. */
      return is_fahr ? cels(temp) : fahr(temp);
}
```

8 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.

107) 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. Similarly, the address of the function is not affected by the function’s being inlined.

108) For example, an implementation might never perform inline substitution, or might only perform inline substitutions to calls in the scope of an **inline** declaration.

109) Since an inline definition is distinct from the corresponding external definition, and from any other corresponding inline definition in another translation unit, all corresponding objects with static storage duration are also distinct in each of the definitions.

6.7.5 Declarators

Syntax

1 *declarator:*
  pointer$_{opt}$ *direct-declarator*

direct-declarator:
  identifier
  (*declarator* **)**
  direct-declarator **[** *assignment-expression*$_{opt}$ **]**
  direct-declarator **[** ***** **]**
  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$_{opt}$

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 the nested sequence of declarators in a full declarator contains a variable length array type, the type specified by the full declarator is said to be *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 incomplete type, either directly or via one or more **`typedef`**s.

Forward references: array declarators (6.7.5.2), type definitions (6.7.7).

6.7.5.1 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**''.

6.7.5.2 Array declarators

Constraints

1 The **[** and **]** may delimit an expression or *****. If **[** and **]** delimit an expression (which specifies the size of an array), it shall have an integer type. If the expression is a constant expression then it shall have a value greater than zero. The element type shall not be an incomplete or function type.

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

Semantics

3 If, in the declaration ''**T D1**'', **D1** has the form

D[*assignment-expression*$_{opt}$**]**

or

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* array of *T*''.[110] If the size is not present, the array type is an incomplete type. If ***** is used instead of a size expression, the array type is a variable length array type of unspecified size, which can only be used in declarations with function prototype scope.[111] If the size expression 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. If the size expression is not a constant expression, and it is evaluated at program execution time, it shall evaluate to a value greater than zero. It is unspecified whether side effects are produced when the size expression is evaluated. The size of each instance of a variable length array type does not change during its lifetime.

4 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.

---

110) When several ''array of'' specifications are adjacent, a multidimensional array is declared.

111) Thus, ***** can be used only in function declarations that are not definitions (see 6.7.5.3).

5 EXAMPLE 1

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

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

6 EXAMPLE 2 Note the distinction between the declarations

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

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

7 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; // Error - not compatible because 4 != 6.
        r = c; // Compatible, but defined behavior
               // only if n == 6 and m == n+1.
}
```

8 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 **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];                       // Error - file scope VLA.
extern int (*p2)[n];            // Error - file scope VM.
int B[100];                     // OK - file scope but not VM.

void fvla(int m, int C[m][m])   // OK - VLA with prototype scope.
{
      typedef int VLA[m][m]     // OK - block scope typedef VLA.

      struct tag {
            int (*y)[n];        // Error - y not ordinary identifier.
            int z[n];           // Error - z not ordinary identifier.
      };
      int D[m];                 // OK - auto VLA.
      static int E[m];          // Error - static block scope VLA.
      extern int F[m];          // Error - F has linkage and is VLA.
      int (*s)[m];              // OK - auto pointer to VLA.
      extern int (*r)[m];       // Error - r had linkage and is
                                // a pointer to VLA.
      static int (*q)[m] = &B;  // OK - q is a static block
                                // pointer to VLA.
}
```

Forward references: function declarators (6.7.5.3), function definitions (6.9.1), initialization (6.7.8).

6.7.5.3 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.[112)]

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

112) Arrays and functions are rewritten as pointers.

T''.

6 A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function. A declaration of a parameter as ''array of *type*'' shall be adjusted to ''pointer to *type*'', and a declaration of a parameter as ''function returning *type*'' shall be adjusted to ''pointer to function returning *type*'', as in 6.3.2.1. If the list terminates with an ellipsis (`, ...`), no information about the number or types of the parameters after the comma is supplied.[113] The special case of an unnamed parameter of type **`void`** as the only item in the list specifies that the function has no parameters.

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

8 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.

9 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.

10 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.[114]

11 For two function types to be compatible, both shall specify compatible return types.[115] 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.)

113) The macros defined in the **`<stdarg.h>`** header (7.15) may be used to access arguments that correspond to the ellipsis.

114) See ''future language directions'' (6.11.3).

115) If both function types are ''old style'', parameter types are not compared.

12 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 designator, which is then used to call the function; it returns an **int**.

13 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.

14 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**.

15 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.

16 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;
}
```

17 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]);
```

Forward references: function definitions (6.9.1), type names (6.7.6).

6.7.6 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$_{opt}$ **[** *assignment-expression*$_{opt}$ **]**
  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.[116)]

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 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**.

6.7.7 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 6.7.5. 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

116) 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.

```
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 at least the range [−15, +15], and one named **r** that contains values in the range [0, 31] or values in at least the range [−15, +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];
}
```

Forward references: the **`signal`** function (7.14.1.1).

6.7.8 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 an object type that is not a variable length array type.

4 All the expressions in an initializer for an object that has static 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 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;

— if it is a union, the first named member is initialized (recursively) according to these rules.

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, optionally enclosed in braces. Successive characters of the character 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 **wchar_t** may be initialized by a wide string literal, 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.[117] 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.[118]

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.[119] 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; all subobjects that are not initialized explicitly shall be initialized implicitly the same as objects that have static storage duration.

20 If the aggregate contains elements or members that are aggregates or unions, or if the first member of a union is an aggregate or union, 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 first member of 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

117) 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.

118) 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.

119) 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.

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.

22 If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. At the end of its initializer list, the array no longer has incomplete type.

23 The order in which any side effects occur among the initialization list expressions is unspecified.[120)]

24 EXAMPLE 1 Provided that **<complex.h>** has been **#include**d, the declarations

```
int i = 3.5;
complex c = 5 + 3 * I;
```

define and initialize **i** with the value 3 and **c** with the value $5.0 + 3.0i$.

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

120) In particular, the evaluation order need not be the same as the order of subobject initialization.

```
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 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>** (7.17).

6.8 Statements

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 *full expression* is an expression that is not part of another expression or declarator. Each of the following is a full expression: an initializer; 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. The end of a full expression is a sequence point.

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.

Semantics

3 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, or block

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* (also called 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 that 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.

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.[121)]

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);
```

5 EXAMPLE 2 In the program fragment

121) Such as assignments, and function calls which have side effects.

```
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.

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 the **`switch`** statement causes a jump to within the scope of an identifier with a variably modified type, the entire **`switch`** statement shall be within the scope of that identifier.[122)]

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.

---

122) 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.

Implementation limits

6 As discussed in 5.2.4.1, the implementation may limit the number of **case** values in a **switch** statement.

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.

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 Except for the behavior of a **`continue`** statement in the loop body, the statement

`for (` *clause-1* **`;`** *expression-2* **`;`** *expression-3* **`)`** *statement*

and the sequence of statements

```
{
        clause-1 ;
        while ( expression-2 ) {
                statement
                expression-3 ;
        }
}
```

are equivalent (where *clause-1* can be an expression or a declaration).[123] Unlike the other iteration statements, the **`for`** statement introduces new blocks that limit the scope of declarations and compound literals occurring in the loop.

2 Both *clause-1* and *expression-3* can be omitted. If *clause-1* is an expression, it is evaluated as a void expression, as is *expression-3*. An omitted *expression-2* is replaced by a nonzero constant.

Forward references: the **`continue`** statement (6.8.6.2).

123) Thus, *clause-1* specifies initialization for the loop, possibly declaring one or more variables for use in the loop; *expression-2*, the controlling expression, specifies an evaluation made before each iteration, such that execution of the loop continues until the expression compares equal to 0; *expression-3* specifies an operation (such as incrementing) that is performed after each iteration. If *clause-1* is a declaration, then the scope of any variable it declares is the remainder of the declaration and the entire loop, including the other two expressions.

6.8.6 Jump statements

Syntax

1 *jump-statement:*
  goto *identifier* **;**
  continue ;
  break ;
  return *expression*$_{opt}$ **;**

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;              // Error: going INTO scope of VLA.
{
      double a[n];
      a[j] = 4.4;
lab3:
      a[j] = 3.3;
      goto lab4;        // OK, going WITHIN scope of VLA.
      a[j] = 5.5;
lab4:
      a[j] = 6.6;
}
goto lab4;              // Error: 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;**.[124)]

124) Following the **contin:** label is a null statement.

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.

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.[125)] *

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 |

---

125) 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.

a function call to fetch the value).

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`** operator), 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 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`** operator), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.[126)]

6.9.1 Function definitions

Syntax

1 *function-definition:*
    declaration-specifiers declarator declaration-list$_{opt}$ *compound-statement*

declaration-list:
    declaration
    declaration-list declaration

126) Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.

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.[127)]

3 The return type of a function shall be **void** or an 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.

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,[128)] 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 6.7.5.3 for a parameter type list; the resulting type shall be an 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.

---

127) 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 of no arguments 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 */
```

128) See ‘‘future language directions’’ (6.11.4).

9 Each parameter has automatic storage duration. Its identifier is an lvalue, which is in effect declared at the head of the compound statement that constitutes the function body (and therefore cannot be redeclared in the function body except in an enclosed block). The layout of the storage for parameters is unspecified.

10 On entry to the function, all size expressions of variably modified parameters 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:

```
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:
pp-tokens$_{opt}$ *new-line*
if-section
control-line

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*

lparen:
a **(** character not immediately preceded by white-space

replacement-list:
$pp\text{-}tokens_{opt}$

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 begins with 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, and is ended by the next new-line character.[129)] A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

Constraints

3 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

4 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.

5 The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

6 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.

129) 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).

6.10.1 Conditional inclusion

Constraints

1 The expression that controls conditional inclusion shall be an integer constant expression except that: it shall not contain a cast; identifiers (including those lexically identical to keywords) are interpreted as described below;[130] 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 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.

Semantics

2 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.

3 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 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, except that 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>`**. 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.[131] Also, whether a single-character character constant may have a negative value is implementation-defined.

4 Preprocessing directives of the forms

130) 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.

 # 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.

5 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.[132)]

Forward references: macro replacement (6.10.3), source file inclusion (6.10.2), largest integer types (7.18.1.5).

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

131) 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)
```

132) 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.[133] 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 letters or digits (as defined in 5.2.1) followed by a period (**.**) and a single letter. The first character shall be a letter. The implementation may ignore the 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"
```

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
```

133) 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.

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 a macro without use of *lparen* (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.

3 An identifier currently defined as a macro using *lparen* (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.

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 agree with 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 only occur in the replacement-list of a **`#define`** preprocessing directive using the ellipsis notation in the arguments.

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[134] to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. *

134) 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.

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 arguments, 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, 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: 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 unspecified 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.

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.

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. Then, the resulting preprocessing token sequence is 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. Further, 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.

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 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 // from previous #include example
#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 **##** *placemarker*

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)[135] causes the implementation to behave in a manner which it shall document. 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 whose meanings are described elsewhere:

---

135) 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.

#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).

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 following macro names shall be defined by the implementation:

__LINE__ The presumed line number (within the current source file) of the current source line (a decimal constant).[136)]

__FILE__ The presumed name of the current source file (a character string literal).[136)]

__DATE__ The date of translation of the source file: 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.

__TIME__ The time of translation of the source file: 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.

__STDC__ The decimal constant 1, intended to indicate a conforming implementation.

__STDC_VERSION__ The decimal constant **199901L**.[137)]

2 The following macro names are conditionally defined by the implementation:

__STDC_ISO_10646__ A decimal constant of the form **yyyymmL** (for example, **199712L**), intended to indicate that values of type **wchar_t** are the coded representations of the characters

136) The presumed line number and source file name can be changed by the **#line** directive.

137) This macro was not specified in ISO/IEC 9899:1990 and was specified as **199409L** in ISO/IEC 9899:AMD1:1995

defined by ISO/IEC 10646, along with all amendments and technical corrigenda as of the specified year and month.

`__STDC_IEC_559__` The decimal constant 1, intended to indicate conformance to the specifications in annex F (IEC 60559 floating-point arithmetic).

`__STDC_IEC_559_COMPLEX__` The decimal constant 1, intended to indicate adherence to the specifications in informative annex G (IEC 60559 compatible complex arithmetic).

3 The values of the predefined macros (except for **`__LINE__`** and **`__FILE__`**) remain constant throughout the translation unit.

4 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.

Forward references: the **`asctime`** function (7.23.3.1).

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 the **`L`** prefix, if present, 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 Character escape sequences

1 Lowercase letters as escape sequences are reserved for future standardization. Other characters may be used in extensions.

6.11.2 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.

6.11.3 Function declarators

1 The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.

6.11.4 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.

6.11.5 Pragma directives

1 Pragmas whose first *pp-token* is **`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 characters preceding the null character and the *value of a string* is the sequence of the values of the contained characters, in order.

2 A *letter* is a printing character in the execution character set corresponding to any of the 52 required lowercase and uppercase letters in the source character set, listed in 5.2.1.

3 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.[138] It is represented in the text and examples by a period, but may be changed by the **setlocale** function.

4 A *wide character* is a code value (a binary encoded integer) of an object of type **wchar_t** that corresponds to a member of the extended character set.[139]

5 A *null wide character* is a wide character with code value zero.

6 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.

7 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.[140]

---

138) The functions that make use of the decimal-point character are the string conversion functions (7.20.1), the wide-string numeric conversion functions (7.24.4.1), the formatted input/output functions (7.19.6), and the formatted wide-character input/output functions (7.24.2).

139) An equivalent definition can be found in 6.4.4.4.

140) 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.

Forward references: character handling (7.4), the **setlocale** function (7.11.1.1).

7.1.2 Standard headers

1 Each library function is declared, with a type that includes a prototype, in a *header*,[141] 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

<assert.h>	**<inttypes.h>**	**<signal.h>**	**<stdlib.h>**
<complex.h>	**<iso646.h>**	**<stdarg.h>**	**<string.h>**
<ctype.h>	**<limits.h>**	**<stdbool.h>**	**<tgmath.h>**
<errno.h>	**<locale.h>**	**<stddef.h>**	**<time.h>**
<fenv.h>	**<math.h>**	**<stdint.h>**	**<wchar.h>**
<float.h>	**<setjmp.h>**	**<stdio.h>**	**<wctype.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.

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.

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).

141) A header is not necessarily a source file, nor are the **<** and **>** delimited sequences in header names necessarily valid source file names.

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) are always reserved for use as identifiers with external linkage.[142)]

— Each identifier with file scope listed in any of the following subclauses (including the future library directions) is reserved for use as macro 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.

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 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

142) The list of reserved identifiers with external linkage includes **`errno`**, **`setjmp`**, and **`va_end`**.

permitted to take the address of a library function even if it is also defined as a macro.[143)] 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.[144)] 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.[145)] All object-like macros listed as expanding to 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 storage duration.[146)]

5 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);
```

143) This means that an implementation shall provide an actual function for each library function, even if it also provides a macro for that function.

144) Such macros might not contain the sequence points that the corresponding function calls do.

145) 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.

146) Thus, a signal handler cannot, in general, call standard library functions.

— by use of its associated header (assuredly generating a true function reference)

```
#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** macro 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.

7.2.1 Program diagnostics

7.2.1.1 The **assert** macro

Synopsis

1
```
#include <assert.h>
void assert(_Bool expression);
```

Description

2 The **assert** macro puts diagnostic tests into programs. When it is executed, if **expression** 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 file in an implementation-defined format.[147] It then calls the **abort** function.

Returns

3 The **assert** macro returns no value.

Forward references: the **abort** function (7.20.4.1).

147) The message written might be of the form:

Assertion failed: *expression*, **function** *abc*, **file** *xyz*, **line** *nnn* .

7.3 Complex arithmetic <complex.h>

7.3.1 Introduction

1 The header **<complex.h>** defines macros and declares functions that support complex arithmetic.[148] 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 same name but with `f` and `l` suffixes which are corresponding functions with `float` and `long double` parameters and return values.

2 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.[149]

3 The macros

    imaginary

and

    _Imaginary_I

are defined if and only if the implementation supports imaginary types;[150] if defined, they expand to `_Imaginary` and a constant expression of type `const float _Imaginary` with the value of the imaginary unit.

4 The macro

    I

expands to either `_Imaginary_I` or `_Complex_I`. If `_Imaginary_I` is not defined, `I` shall expand to `_Complex_I`.

5 Notwithstanding the provisions of 7.1.3, a program is permitted to undefine and perhaps then redefine the macros `complex`, `imaginary`, and `I`.

Forward references: IEC 60559-compatible complex arithmetic (annex G).

---

148) See "future library directions" (7.26.1).

149) The imaginary unit is a number i such that $i^2 = -1$.

150) A specification for imaginary types is in informative 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 specification 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 formula 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.[151] 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

151) 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.

another **CX_LIMITED_RANGE** pragma is encountered (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 $[-\pi/2, \pi/2]$ 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`** function computes 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 non-negative 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 range $[-\pi, \pi]$.

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**.[152)]

Returns

3 The **cimag** functions return the imaginary part value (as a real).

152) For a variable **z** of complex type, `z == creal(z) + cimag(z)*I`.

7.3.9.3 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.

7.3.9.4 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.

7.3.9.5 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`**.[153)]

Returns

3 The **`creal`** functions return the real part value.

153) 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 testing and mapping characters.[154] 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.[155]

Forward references: **EOF** (7.19.1), localization (7.11).

7.4.1 Character testing 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.

154) See ‘‘future library directions’’ (7.26.2).

155) 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).

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 none of **iscntrl**, **isdigit**, **ispunct**, or **isspace** is true.[156] In the **"C"** locale, **isalpha** returns true only for the characters for which **isupper** or **islower** is true.

7.4.1.3 The `iscntrl` function

Synopsis

1
```
#include <ctype.h>
int iscntrl(int c);
```

Description

2 The **iscntrl** function tests for any control character.

7.4.1.4 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).

156) The functions **islower** and **isupper** test true or false separately for each of these additional characters; all four combinations are possible.

7.4.1.5 The `isgraph` function

Synopsis

1
```
#include <ctype.h>
int isgraph(int c);
```

Description

2 The `isgraph` function tests for any printing character except space (`' '`).

7.4.1.6 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 characters defined as lowercase letters (as defined in 5.2.1).

7.4.1.7 The `isprint` function

Synopsis

1
```
#include <ctype.h>
int isprint(int c);
```

Description

2 The `isprint` function tests for any printing character including space (`' '`).

7.4.1.8 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.

7.4.1.9 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.

7.4.1.10 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 characters defined as uppercase letters (as defined in 5.2.1).

7.4.1.11 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 6.4.4.2).

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[157] that has type **int**, the value of which is set to a positive error number by several library functions. It is unspecified whether **errno** is a macro or an identifier declared with external linkage. 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** is zero at program startup, but is never set to zero by any library function.[158] 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,[159] may also be specified by the implementation.

---

157) 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()**).

158) 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.

159) See ‘‘future library directions’’ (7.26.3).

7.6 Floating-point environment **<fenv.h>**

1 The header **<fenv.h>** declares two types and several macros and functions to 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.[160] A *floating-point status flag* is a system variable whose value is set as a side effect of floating-point arithmetic to provide auxiliary information. 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 Certain programming conventions support the intended model of use for the floating-point environment:[161]

— a function call does not alter its caller's modes, clear its caller's flags, nor depend on the state of its caller's flags unless the function is so documented;

— a function call is assumed to require default modes, unless its documentation promises otherwise or unless the function is known not to use floating-point;

— a function call is assumed to have the potential for raising floating-point exceptions, unless its documentation promises otherwise, or unless the function is known not to use floating-point.

3 The type

```
fenv_t
```

represents the entire floating-point environment.

4 The type

```
fexcept_t
```

represents the floating-point exception flags collectively, including any status the implementation associates with the flags.

5 Each of the macros

```
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
```

is defined if and only if the implementation supports the exception by means of the functions in 7.6.2. Additional floating-point exceptions, with macro definitions beginning

160) This header is designed to support the exception status flags and directed-rounding control modes required by IEC 60559, and other similar floating-point state information. Also it is designed to facilitate code portability among all systems.

161) With these conventions, a programmer can safely assume default 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.

with **FE_** and an uppercase letter, 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.

6 The macro

FE_ALL_EXCEPT

is simply the bitwise OR of all exception macros defined by the implementation.

7 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 rounding directions, with macro definitions beginning with **FE_** and an uppercase letter, may also be specified by the implementation. The defined macros expand to integer constant expressions whose values are distinct nonnegative values.[162)]

8 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.

9 Additional macro definitions, beginning with **FE_** and having type *pointer to const-qualified* **fenv_t**, may also be specified by the implementation.

162) Even though the rounding direction macros may expand to constants corresponding to the values of **FLT_ROUNDS**, they are not required to do so.

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 flags or run under non-default modes.[163)] 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 (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 flags or runs under non-default mode settings, but was translated with the state for the **FENV_ACCESS** pragma *off*, then the behavior is undefined. The default state (*on* or *off*) for the pragma is implementation-defined.

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**.[164)]

---

163) The purpose of the **FENV_ACCESS** pragma is to allow certain optimizations, for example *global common subexpression elimination*, *code motion*, and *constant folding*, that could subvert flag tests and mode changes. In general, if the state of **FENV_ACCESS** is *off* then the translator can assume that default modes are in effect and the flags are not tested.

164) 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.

7.6.2 Exceptions

1 The following functions provide access to the exception flags.[165] 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 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>
void feclearexcept(int excepts);
```

Description

2 The **feclearexcept** function clears the supported exceptions represented by its argument.

7.6.2.2 The fegetexceptflag function

Synopsis

1
```
#include <fenv.h>
void fegetexceptflag(fexcept_t *flagp,
     int excepts);
```

Description

2 The **fegetexceptflag** function stores an implementation-defined representation of the exception flags indicated by the argument **excepts** in the object pointed to by the argument **flagp**.

165) The functions **fetestexcept**, **feraiseexcept**, and **feclearexcept** support the basic abstraction of flags that are either set or clear. An implementation may endow exception flags with more information — for example, the address of the code which first raised the exception; the functions **fegetexceptflag** and **fesetexceptflag** deal with the full content of flags.

7.6.2.3 The `feraiseexcept` function

Synopsis

1
```
#include <fenv.h>
void feraiseexcept(int excepts);
```

Description

2 The **feraiseexcept** function raises the supported exceptions represented by its argument.[166] The order in which these exceptions are raised is unspecified, except as stated in F.7.6. Whether the **feraiseexcept** function additionally raises the *inexact* exception whenever it raises the *overflow* or *underflow* exception is implementation-defined.

7.6.2.4 The `fesetexceptflag` function

Synopsis

1
```
#include <fenv.h>
void fesetexceptflag(const fexcept_t *flagp,
     int excepts);
```

Description

2 The **fesetexceptflag** function sets the complete status for those exception flags indicated by the argument **excepts**, according to the representation 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 exceptions represented by the argument **excepts**. This function does not raise exceptions, but only sets the state of the flags.

166) The effect is intended to be similar to that of exceptions raised by arithmetic operations. Hence, enabled traps for exceptions raised by this function are taken. The specification in F.7.6 is in the same spirit.

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 exception flags are currently set. The **excepts** argument specifies the exception flags to be queried.[167)]

Returns

3 The **fetestexcept** function returns the value of the bitwise OR of the exception macros corresponding to the currently set exceptions included in **excepts**.

4 EXAMPLE Call **f** if *invalid* is set, then **g** if *overflow* is set:

```
#include <fenv.h>
/* ... */
{
      #pragma STDC FENV_ACCESS ON
      int set_excepts;
      // 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.

167) This mechanism allows testing several exceptions with just one function call.

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.

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 a zero value if and only if the argument is equal to a rounding direction macro (that is, if and only if the requested rounding direction can be established).

4 EXAMPLE 1 Save, set, and restore the rounding direction. Report an error and abort if setting the rounding direction fails.

```
#include <fenv.h>
#include <assert.h>
/* ... */
{
      #pragma STDC FENV_ACCESS ON
      int save_round;
      int setround_ok;
      save_round = fegetround();
      setround_ok = fesetround(FE_UPWARD);
      assert(setround_ok);
      /* ... */
      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>
void fegetenv(fenv_t *envp);
```

Description

2 The **fegetenv** function stores the current floating-point environment in the object pointed to by **envp**.

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 exception flags, and then installs a *non-stop* (continue on exceptions) mode, if available, for all exceptions.[168)]

Returns

3 The **feholdexcept** function returns zero if and only if non-stop exception handling was successfully installed.

168) 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 exceptions from their callers.

7.6.4.3 The `fesetenv` function

Synopsis

1
```
#include <fenv.h>
void fesetenv(const fenv_t *envp);
```

Description

2 The **fesetenv** function establishes 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 the macro **FE_DFL_ENV** or an implementation-defined environment macro. Note that **fesetenv** merely installs the state of the exception flags represented through its argument, and does not raise these exceptions.

7.6.4.4 The `feupdateenv` function

Synopsis

1
```
#include <fenv.h>
void feupdateenv(const fenv_t *envp);
```

Description

2 The **feupdateenv** function saves the currently raised exceptions in its automatic storage, installs the floating-point environment represented by the object pointed to by **envp**, and then raises the saved exceptions. The argument **envp** shall point to an object set by a call to **feholdexcept** or **fegetenv**, or equal the macro **FE_DFL_ENV** or an implementation-defined environment macro.

3 EXAMPLE 1 Hide spurious underflow exceptions:

```
#include <fenv.h>
double f(double x)
{
      #pragma STDC FENV_ACCESS ON
      double result;
      fenv_t save_env;
      feholdexcept(&save_env);
      // compute result
      if (/* test spurious underflow */)
            feclearexcept(FE_UNDERFLOW);
      feupdateenv(&save_env);
      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 four functions for converting numeric character strings to greatest-width integers and, for each type declared in **<stdint.h>**, it defines corresponding macros for conversion specifiers for use with the formatted input/output functions.[169]

Forward references: integer types **<stdint.h>** (7.18).

7.8.1 Macros for format specifiers

1 Each of the following object-like macros[170] 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** family) or **SCN** (character string literals for the **fscanf** family),[171] followed by the conversion specifier, followed by a name corresponding to a similar type name in 7.18.1. 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:

PRId8	**PRId16**	**PRId32**	**PRId64**
PRIdLEAST8	**PRIdLEAST16**	**PRIdLEAST32**	**PRIdLEAST64**
PRIdFAST8	**PRIdFAST16**	**PRIdFAST32**	**PRIdFAST64**
PRIdMAX	**PRIdPTR**		

PRIi8	**PRIi16**	**PRIi32**	**PRIi64**
PRIiLEAST8	**PRIiLEAST16**	**PRIiLEAST32**	**PRIiLEAST64**
PRIiFAST8	**PRIiFAST16**	**PRIiFAST32**	**PRIiFAST64**
PRIiMAX	**PRIiPTR**		

3 The **fprintf** macros for unsigned integers are:

PRIo8	**PRIo16**	**PRIo32**	**PRIo64**
PRIoLEAST8	**PRIoLEAST16**	**PRIoLEAST32**	**PRIoLEAST64**
PRIoFAST8	**PRIoFAST16**	**PRIoFAST32**	**PRIoFAST64**
PRIoMAX	**PRIoPTR**		

169) See “future library directions” (7.26.4).

170) C++ implementations should define these macros only when **__STDC_FORMAT_MACROS** is defined before **<inttypes.h>** is included.

171) 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.

PRIu8	**PRIu16**	**PRIu32**	**PRIu64**
PRIuLEAST8	**PRIuLEAST16**	**PRIuLEAST32**	**PRIuLEAST64**
PRIuFAST8	**PRIuFAST16**	**PRIuFAST32**	**PRIuFAST64**
PRIuMAX	**PRIuPTR**		
PRIx8	**PRIx16**	**PRIx32**	**PRIx64**
PRIxLEAST8	**PRIxLEAST16**	**PRIxLEAST32**	**PRIxLEAST64**
PRIxFAST8	**PRIxFAST16**	**PRIxFAST32**	**PRIxFAST64**
PRIxMAX	**PRIxPTR**		
PRIX8	**PRIX16**	**PRIX32**	**PRIX64**
PRIXLEAST8	**PRIXLEAST16**	**PRIXLEAST32**	**PRIXLEAST64**
PRIXFAST8	**PRIXFAST16**	**PRIXFAST32**	**PRIXFAST64**
PRIXMAX	**PRIXPTR**		

4 The **fscanf** macros for signed integers are:

SCNd8	**SCNd16**	**SCNd32**	**SCNd64**
SCNdLEAST8	**SCNdLEAST16**	**SCNdLEAST32**	**SCNdLEAST64**
SCNdFAST8	**SCNdFAST16**	**SCNdFAST32**	**SCNdFAST64**
SCNdMAX	**SCNdPTR**		
SCNi8	**SCNi16**	**SCNi32**	**SCNi64**
SCNiLEAST8	**SCNiLEAST16**	**SCNiLEAST32**	**SCNiLEAST64**
SCNiFAST8	**SCNiFAST16**	**SCNiFAST32**	**SCNiFAST64**
SCNiMAX	**SCNiPTR**		

5 The **fscanf** macros for unsigned integers are:

SCNo8	**SCNo16**	**SCNo32**	**SCNo64**
SCNoLEAST8	**SCNoLEAST16**	**SCNoLEAST32**	**SCNoLEAST64**
SCNoFAST8	**SCNoFAST16**	**SCNoFAST32**	**SCNoFAST64**
SCNoMAX	**SCNoPTR**		
SCNu8	**SCNu16**	**SCNu32**	**SCNu64**
SCNuLEAST8	**SCNuLEAST16**	**SCNuLEAST32**	**SCNuLEAST64**
SCNuFAST8	**SCNuFAST16**	**SCNuFAST32**	**SCNuFAST64**
SCNuMAX	**SCNuPTR**		
SCNx8	**SCNx16**	**SCNx32**	**SCNx64**
SCNxLEAST8	**SCNxLEAST16**	**SCNxLEAST32**	**SCNxLEAST64**
SCNxFAST8	**SCNxFAST16**	**SCNxFAST32**	**SCNxFAST64**
SCNxMAX	**SCNxPTR**		

6 Because the default argument promotions do not affect pointer parameters, there might not exist suitable **fscanf** format specifiers for some of the types defined in this header. Consequently, as a special exception to the requirement that the implementation define all macros associated with each type defined by this header, in such a case the problematic **fscanf** macros may be left undefined.

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 Conversion functions for greatest-width integer types

7.8.2.1 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**.

7.8.2.2 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**.

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 7.17); 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.[172] Additional macro definitions, beginning with the characters **LC_** and an uppercase letter,[173] 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[174] 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 **strfxtime** 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**.

4 At program startup, the equivalent of

```
setlocale(LC_ALL, "C");
```

is executed.

5 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.

---

172) ISO/IEC 9945−2 specifies locale and charmap formats that may be used to specify locales for C.

173) See ‘‘future library directions’’ (7.26.5).

174) The only functions in 7.4 whose behavior is not affected by the current locale are **isdigit** and **isxdigit**.

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.[175)]

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 (7.19.6), the multibyte character functions (7.20.7), the multibyte string functions (7.20.8), string conversion functions (7.20.1), the **strcoll** function (7.21.4.3), the **strftime** function (7.23.3.5), the **strfxtime** function (7.23.3.6), the **strxfrm** function (7.21.4.5).

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.

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. *

175) The implementation shall arrange to encode in a string the various categories due to a heterogeneous locale when **category** has the value **LC_ALL**.

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 4217:1995. 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_currency_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_currency_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_currency_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_currency_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` A space separates the currency symbol and value.

`2` A space separates the currency symbol and the sign string, if adjacent.

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 The following table illustrates the rules which may well be used by four countries to format monetary quantities.

	Local format		International format	
Country	Positive	Negative	Positive	Negative
Finland	**`1.234,56 mk`**	**`-1.234,56 mk`**	**`FIM 1.234,56`**	**`FIM -1.234,56`**
Italy	**`L.1.234`**	**`-L.1.234`**	**`ITL 1.234`**	**`-ITL 1.234`**
Netherlands	**`ƒ 1.234,56`**	**`ƒ -1.234,56`**	**`NLG 1.234,56`**	**`NLG -1.234,56`**
Switzerland	**`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`** are:

	Finland	Italy	Netherlands	Switzerland
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**	**1**	**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	**1**	**1**	**1**	**1**
int_p_sign_posn	**1**	**1**	**1**	**1**
int_n_sign_posn	**4**	**1**	**4**	**2**

7.12 Mathematics <math.h>

1 The header <math.h> declares two types and several 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.[176] 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.[177]

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`.[178]

4 The macro

```
INFINITY
```

expands to a constant expression of type `float` representing an implementation-defined positive or unsigned infinity, if available; else to a positive constant of type `float` that overflows at translation time.[179]

---

176) 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.

177) 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.

178) `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL` can be positive infinities in an implementation that supports infinities.

179) In this case, using `INFINITY` will violate the constraint in 6.4.4 and thus require a diagnostic.

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 an implementation-defined quiet NaN.

6 The macros

```
FP_INFINITE
FP_NAN
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
```

are for number classification. They represent the mutually exclusive kinds of floating-point values. They expand to integer constant expressions with distinct values.

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.[180] The macros

```
FP_FAST_FMAF
FP_FAST_FMAL
```

are, respectively, **`float`** and **`long double`** analogs of **`FP_FAST_FMA`**.

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`**. *

Recommended practice

9 Conversion from (at least) **`double`** to decimal with **`DECIMAL_DIG`** digits and back should be the identity function (which assures that conversion from the widest supported IEC 60559 format to decimal with **`DECIMAL_DIG`** digits and back is the identity function).

180) 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.

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.

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.[181] On a domain error, the function returns an implementation-defined value; whether the integer expression **errno** acquires the value **EDOM** is implementation-defined.

3 Similarly, 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. 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, or if the mathematical result is an exact infinity (for example **log(0.0)**), 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; whether **errno** acquires the value **ERANGE** when a range error occurs is implementation-defined. 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.[182] If the result underflows, the function returns a value whose magnitude is no greater than the smallest normalized positive number in the specified type and is otherwise implementation-defined; whether **errno** acquires the value **ERANGE** is implementation-defined.

---

181) 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.

182) The term underflow here is intended to encompass both *gradual underflow* as in IEC 60559 and also *flush-to-zero* underflow.

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 (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. The result is undefined if an argument is not 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, or zero. 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.[183)]

Returns

3 The **`fpclassify`** macro returns the value of the number classification macro appropriate to the value of its argument.

4 EXAMPLE The **`fpclassify`** macro might be implemented in terms of ordinary functions as

---

183) 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`**.

```
#define fpclassify(x) \
      ((sizeof (x) == sizeof (float)) ? \
            __fpclassifyf(x) \
      : (sizeof (x) == sizeof (double)) ? \
            __fpclassifyd(x) \
      :     __fpclassifyl(x))
```

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.[184)]

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.

---

184) 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.

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.[185]

Returns

3 The **signbit** macro returns a nonzero value if and only if the sign of its argument value is negative.

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 range [−1, +1].

Returns

3 The **acos** functions return the arc cosine in the range [0, π] 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 range [−1, +1].

Returns

185) The **signbit** macro reports the sign of all values, including infinities, zeros, and NaNs.

3 The **asin** functions return the arc sine in the range $[-\pi/2, +\pi/2]$ 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 the arc tangent in the range $[-\pi/2, +\pi/2]$ 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 principal 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 the arc tangent of **y / x**, in the range $[-\pi, +\pi]$ 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 the cosine value.

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 the sine value.

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 the tangent value.

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 the arc hyperbolic cosine in the range $[0, +\infty]$.

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 the arc hyperbolic sine value.

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 range $[-1, +1]$. A range error may occur if the argument equals -1 or $+1$.

Returns

3 The `atanh` functions return the arc hyperbolic tangent value.

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 the hyperbolic cosine value.

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 the hyperbolic sine value.

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 the hyperbolic tangent value.

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`**: $e^{\mathbf{x}}$. A range error occurs if the magnitude of **`x`** is too large.

Returns

3 The **`exp`** functions return the exponential value.

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`**: $2^{\mathbf{x}}$. A range error occurs if the magnitude of **`x`** is too large.

Returns

3 The **`exp2`** functions return the base-2 exponential value.

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: $e^{\mathbf{x}} - 1$. |
A range error occurs if **`x`** is too large.[186] *

Returns

3 The **`expm1`** functions return the value of $e^{\mathbf{x}} - 1$.

186) For small magnitude **`x`**, **`expm1(x)`** is expected to be more accurate than **`exp(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 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.

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 range error may occur if **`x`** is 0.

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 the value of $\mathbf{x} \times 2^{\mathbf{exp}}$.

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 range error may occur if the argument is zero.

Returns

3 The **log** functions return the base-*e* logarithm value.

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 range error may occur if the argument is zero.

Returns

3 The **log10** functions return the base-10 logarithm value.

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.[187] A domain error occurs if the argument is less than −1. A range error may occur if the argument equals −1.

Returns

3 The **log1p** functions return the value of the base-*e* logarithm of 1 plus the argument.

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 range error may occur if the argument is zero.

Returns

3 The **log2** functions return the base-2 logarithm value.

187) For small magnitude **x**, **log1p(x)** is expected to be more accurate than **log(1 + 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 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 value of 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 the value of $\mathbf{x} \times \mathbf{FLT\_RADIX}^n$. *

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 the value of the cube root.

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 the absolute value of **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 the value of the square root of the sum of the squares.

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 negative and **y** is finite and not an integer value. A domain error occurs if the result cannot be represented when **x** is zero and **y** is less than or equal to zero. A range error may occur.

Returns

3 The **pow** functions return the value of **x** raised to the power **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 the value of the square root.

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**: $\frac{2}{\sqrt{\pi}} \int_0^{\mathbf{x}} e^{-t^2}\,dt$.

Returns

3 The **erf** functions return the error function value.

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**: $\frac{2}{\sqrt{\pi}} \int_{\mathbf{x}}^{\infty} e^{-t^2}\,dt$. A range error occurs if **x** is too large.

Returns

3 The **erfc** functions return the complementary error function value.

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**: $\log_e |\Gamma(\mathbf{x})|$. A range error occurs if **x** is too large or if **x** is a negative integer or zero.

Returns

3 The **lgamma** functions return the value of the natural logarithm of the absolute value of gamma of **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**: $\Gamma(\mathbf{x})$. A domain error occurs if **x** is a negative integer or zero. A range error may occur if the magnitude of **x** is too large or too small.

Returns

3 The **tgamma** functions return the gamma function value.

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`**: $\lceil \mathbf{x} \rceil$.

Returns

3 The **`ceil`** functions return the smallest integer value not less than **`x`**, 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`**: $\lfloor \mathbf{x} \rfloor$.

Returns

3 The **`floor`** functions return the largest integer value not greater than **`x`**, 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* 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 do raise the inexact exception if the result differs in value from the argument (see F.9.6.3 and F.9.6.4).

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. A range error may occur if the magnitude of **x** is too large.

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. A range error may occur if the magnitude of **x** is too large.

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.[188)]

Returns

3 The **remainder** functions return the value of **x** REM **y**.

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 the value of **x** REM **y**.

7.12.11 Manipulation functions

188) "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. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x." This definition is applicable for all implementations.

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 (7.20.1.3).

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.[189] 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**.

7.12.11.4 The **nextafterx** functions

Synopsis

1
```
#include <math.h>
double nextafterx(double x, long double y);
float nextafterxf(float x, long double y);
long double nextafterxl(long double x, long double y);
```

Description

2 The **nextafterx** functions are equivalent to the **nextafter** functions except that the second parameter has type **long double**.[190]

7.12.12 Maximum, minimum, and positive difference functions

189) The argument values are converted to the type of the function, even by a macro implementation of the function.

190) The result of the **nextafterx** functions is determined in the type of the function, without loss of range or precision in a floating second argument.

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:

$$\begin{array}{ll} x - y & \text{if } x > y \\ +0 & \text{if } x \le y \end{array}$$

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);
```

Description

2 The **fmax** functions determine the maximum numeric value of their arguments.[191)]

Returns

3 The **fmax** functions return the maximum numeric value of their arguments.

191) 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.9.9.2.

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.[192)]

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 the sum **`z`** plus the product **`x`** times **`y`**, rounded as one ternary operation: they computes the sum **`z`** plus the product **`x`** times **`y`** (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value of **`FLT_ROUNDS`**.

Returns

3 The **`fma`** functions return the sum **`z`** plus the product **`x`** times **`y`**, rounded as one ternary operation.

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* exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the *unordered* relationship is true.[193)] The following subclauses

---

192) The **`fmin`** functions are analogous to the **`fmax`** functions in their treatment of NaNs.

193) IEC 60559 requires that the built-in relational operators raise the *invalid* exception if the operands compare unordered, as an error indicator for programs written without consideration of NaNs; the result in these cases is false.

provide macros that are *quiet* (non exception raising) versions of the relational operators, and other comparison macros that facilitate writing efficient code that accounts for NaNs without suffering the *invalid* exception. In the synopses in this subclause, *real-floating* indicates that the argument shall be an expression of real floating type.

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* 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);
```

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* 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* 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* 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* 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.[194]

2 The type declared is

jmp_buf

which is an array type suitable for holding the information needed to restore a calling environment.

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 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**).

194) These functions are useful for dealing with unusual conditions encountered in a low-level function of a program.

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>
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 function containing the invocation of the **setjmp** macro has terminated execution[195] 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 as of the time **longjmp** 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, program 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.

195) 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.

```
#include <setjmp.h>
jmp_buf buf;
void g(int n);
void h(int n);
int n = 6;

void f(void)
{
      int x[n];          // OK, 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,[196] 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.

---

196) See ‘‘future library directions’’ (7.26.9). 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), 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 storage duration 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 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.[197)]

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

197) If any signal is generated by an asynchronous signal handler, the behavior is undefined.

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

is executed for all other signals defined by the implementation.

7 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 (7.20.4.1), the **exit** function (7.20.4.3).

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 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 an 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 (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**.[198)]

7.15.1 Variable argument list access macros

1 The **va_start**, **va_arg**, and **va_copy** macros described in this subclause shall be implemented as macros, not functions. It is unspecified whether **va_end** 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 **va_end**, the behavior is undefined. Each invocation of the **va_start** or **va_copy** macros shall be matched by a corresponding invocation of the **va_end** macro in the function accepting a varying number of arguments.

198) 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.

7.15.1.1 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 be the same as the **va_list ap** initialized by **va_start**. Each invocation of **va_arg** 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.

7.15.1.2 The va_copy macro

Synopsis

1
```
#include <stdarg.h>
void va_copy(va_list dest, va_list src);
```

Description

2 The **va_copy** macro makes the **va_list dest** be a copy of the **va_list src**, as if the **va_start** macro had been applied to it followed by the same sequence of uses of the **va_arg** macro as had previously been used to reach the present state of **src**.

Returns

3 The **va_copy** macro returns no value.

7.15.1.3 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 **va_start** that initialized the **va_list ap**. The **va_end** macro may modify **ap** so that it is no longer usable (without an intervening invocation of **va_start**). If there is no corresponding invocation of the **va_start** 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.

7.15.1.4 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 **va_arg** and **va_end**. **va_start** shall not be invoked again for the same **ap** without an intervening invocation of **va_end** 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 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** shall have visible the definition of the function or a declaration such as

```
void f1(int, ...);
```

7 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, n_ptrs);
      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);
}
```

7.16 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 decimal constant 1,

false

which expands to the decimal constant 0, and

__bool_true_false_are_defined

which expands to the decimal constant 1.

4 Notwithstanding the provisions of 7.1.3, a program is permitted to undefine and perhaps then redefine the macros **bool**, **true**, and **false**.[199)]

199) See ‘‘future library directions’’ (7.26.7).

7.17 Common definitions `<stddef.h>`

1 The following types and macros are defined in the standard header **`<stddef.h>`**. 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; 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 and each member of the basic character set defined in 5.2.1 shall have a code value equal to its value when used as the lone character in an integer character constant.

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.)

Forward references: localization (7.11).

7.18 Integer types `<stdint.h>`

1 The header **`<stdint.h>`** declares sets of integer types having specified widths, and defines corresponding sets of macros.[200] 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 can be declared as a type existing in the implementation,[201] **`<stdint.h>`** shall declare that type, and it shall define the associated macros. Conversely, for each type described herein that cannot be declared as a type existing in the implementation, **`<stdint.h>`** shall not define that type, nor shall it define the associated macros.

7.18.1 Integer types

1 When type 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.

7.18.1.1 Exact-width integer types

1 Each of the following types designates an integer type that has exactly the specified width. These type names have the general form of **`int`***n***`_t`** or **`uint`***n***`_t`** where *n* is the required width. For example, **`uint8_t`** denotes an unsigned integer type that has a width of exactly 8 bits.

2 The following designate exact-width signed integer types:

```
int8_t       int16_t      int32_t      int64_t
```

The following designate exact-width unsigned integer types:

```
uint8_t      uint16_t     uint32_t     uint64_t
```

(These types need not exist in an implementation.)

200) See "future library directions" (7.26.8).

201) Some of these types may denote implementation-defined extended integer types.

7.18.1.2 Minimum-width integer types

1 Each of the following types designates an integer type that has at least the specified width, such that no integer type of lesser size has at least the specified width. These type names have the general form of **int_least***n***_t** or **uint_least***n***_t** where *n* is the minimum required width. For example, **int_least32_t** denotes a signed integer type that has a width of at least 32 bits.

2 The following designate minimum-width signed integer types:

```
int_least8_t          int_least32_t
int_least16_t         int_least64_t
```

The following designate minimum-width unsigned integer types:

```
uint_least8_t         uint_least32_t
uint_least16_t        uint_least64_t
```

(These types exist in all implementations.)

7.18.1.3 Fastest minimum-width integer types

1 Each of the following types designates an integer type that is usually fastest[202] to operate with among all integer types that have at least the specified width. These type names have the general form of **int_fast***n***_t** or **uint_fast***n***_t** where *n* is the minimum required width. For example, **int_fast16_t** denotes the fastest signed integer type that has a width of at least 16 bits.

2 The following designate fastest minimum-width signed integer types:

```
int_fast8_t           int_fast32_t
int_fast16_t          int_fast64_t
```

The following designate fastest minimum-width unsigned integer types:

```
uint_fast8_t          uint_fast32_t
uint_fast16_t         uint_fast64_t
```

(These types exist in all implementations.)

7.18.1.4 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**,

202) 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.

and the result will compare equal to the original pointer:

`uintptr_t`

(These types need not exist in an implementation.)

7.18.1.5 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 exist in all implementations.)

7.18.2 Limits of specified-width integer types

1 The following object-like macros[203)] specify the minimum and maximum limits of the types declared in **`<stdint.h>`**. Each macro name corresponds to a similar type name in 7.18.1.

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.

7.18.2.1 Limits of exact-width integer types

— minimum values of exact-width signed integer types

```
INT8_MIN                          -127
INT16_MIN                       -32767
INT32_MIN                  -2147483647
INT64_MIN         -9223372036854775807
```

(The value shall be either that given or exactly 1 less.)

— maximum values of exact-width signed integer types

```
INT8_MAX                          +127
INT16_MAX                       +32767
INT32_MAX                  +2147483647
INT64_MAX         +9223372036854775807
```

(The value shall be exactly that given.)

203) C++ implementations should define these macros only when **`__STDC_LIMIT_MACROS`** is defined before **`<stdint.h>`** is included.

— maximum values of exact-width unsigned integer types

```
UINT8_MAX                          255
UINT16_MAX                       65535
UINT32_MAX                  4294967295
UINT64_MAX        18446744073709551615
```

(The value shall be exactly that given.)

7.18.2.2 Limits of minimum-width integer types

— minimum values of minimum-width signed integer types

```
INT_LEAST8_MIN                    -127
INT_LEAST16_MIN                 -32767
INT_LEAST32_MIN            -2147483647
INT_LEAST64_MIN   -9223372036854775807
```

— maximum values of minimum-width signed integer types

```
INT_LEAST8_MAX                    +127
INT_LEAST16_MAX                 +32767
INT_LEAST32_MAX            +2147483647
INT_LEAST64_MAX   +9223372036854775807
```

— maximum values of minimum-width unsigned integer types

```
UINT_LEAST8_MAX                    255
UINT_LEAST16_MAX                 65535
UINT_LEAST32_MAX            4294967295
UINT_LEAST64_MAX  18446744073709551615
```

7.18.2.3 Limits of fastest minimum-width integer types

— minimum values of fastest minimum-width signed integer types

```
INT_FAST8_MIN                     -127
INT_FAST16_MIN                  -32767
INT_FAST32_MIN             -2147483647
INT_FAST64_MIN    -9223372036854775807
```

— maximum values of fastest minimum-width signed integer types

```
INT_FAST8_MAX                     +127
INT_FAST16_MAX                  +32767
INT_FAST32_MAX             +2147483647
INT_FAST64_MAX    +9223372036854775807
```

— maximum values of fastest minimum-width unsigned integer types

```
UINT_FAST8_MAX                      255
UINT_FAST16_MAX                   65535
UINT_FAST32_MAX              4294967295
UINT_FAST64_MAX    18446744073709551615
```

7.18.2.4 Limits of integer types capable of holding object pointers

— minimum value of pointer-holding signed integer type

```
INTPTR_MIN                       -32767
```

— maximum value of pointer-holding signed integer type

```
INTPTR_MAX                       +32767
```

— maximum value of pointer-holding unsigned integer type

```
UINTPTR_MAX                       65535
```

7.18.2.5 Limits of greatest-width integer types

— minimum value of greatest-width signed integer type

```
INTMAX_MIN         -9223372036854775807
```

— maximum value of greatest-width signed integer type

```
INTMAX_MAX         +9223372036854775807
```

— maximum value of greatest-width unsigned integer type

```
UINTMAX_MAX        18446744073709551615
```

7.18.3 Limits of other integer types

1 The following object-like macros[204] 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.

— 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*

204) C++ implementations should define these macros only when **__STDC_LIMIT_MACROS** is defined before **<stdint.h>** is included.

— limit of **size_t**

SIZE_MAX	**65535**

— limits of **wchar_t**

WCHAR_MIN	*see below*
WCHAR_MAX	*see below*

— limits of **wint_t**

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** 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.

5 If **wint_t** (see 7.25) 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.

7.18.4 Macros for integer constants

1 The following function-like macros[205)] 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 7.18.1.2 or 7.18.1.5.

2 The argument in any instance of these macros shall be a decimal, octal, or hexadecimal constant (as defined in 6.4.4.1) with a value that does not exceed the limits for the corresponding type.

7.18.4.1 Macros for minimum-width integer constants

1 Each of the following macros expands to an integer constant having the value specified by its argument and a type with at least the specified width. These macro names have the general form of **INT*n*_C** or **UINT*n*_C** where *n* is the minimum required width. For example, **UINT64_C(0x123)** might expand to the integer constant **0x123ULL**.

205) C++ implementations should define these macros only when **__STDC_CONSTANT_MACROS** is defined before **<stdint.h>** is included.

2 The following expand to integer constants that have signed integer types:

INT8_C(*value***)**	**INT32_C(***value***)**
INT16_C(*value***)**	**INT64_C(***value***)**

The following expand to integer constants that have unsigned integer types:

UINT8_C(*value***)**	**UINT32_C(***value***)**
UINT16_C(*value***)**	**UINT64_C(***value***)**

7.18.4.2 Macros for greatest-width integer constants

1 The following macro expands to an integer constant having the value specified by its argument and the type **intmax_t**:

INTMAX_C(*value***)**

The following macro expands to an integer constant having the value specified by its argument and the type **uintmax_t**:

UINTMAX_C(*value***)**

7.19 Input/output **<stdio.h>**

7.19.1 Introduction

1 The header **<stdio.h>** declares three types, several macros, and many functions for performing input and output.

2 The types declared are **size_t** (described in 7.17);

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 an 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 7.17);

_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, which 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;[206]

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 7.19.3.

5 The input/output functions are given the following collective terms:

— The *wide-character input functions* — those functions described in 7.24 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 7.24 that perform output from wide characters and wide strings: **fputwc**, **fputws**, **putwc**, **putwchar**, **fwprintf**, **wprintf**, **vfwprintf**, and **vwprintf**.

— 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**, **gets**, **printf**, **putc**, **putchar**, **puts**, **scanf**, **ungetc**, **vfprintf**, **vfscanf**, **vprintf**, and **vscanf**.

206) 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.

Forward references: files (7.19.3), the **fseek** function (7.19.9.2), streams (7.19.2), the **tmpnam** function (7.19.4.4), **<wchar.h>** (7.24).

7.19.2 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*.[207)]

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 printable 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 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.)[208)]

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:

---

207) 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.

208) The three predefined streams **stdin**, **stdout**, and **stderr** are unoriented at program startup.

— 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.

Environmental limits

7 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 (7.19.5.4), the **`fwide`** function (7.24.3.5), **`mbstate_t`** (7.25.1), the **`fgetpos`** function (7.19.9.1), the **`fsetpos`** function (7.19.9.3).

7.19.3 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 7.19.5.3. 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.[209)]

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.

---

209) 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.

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) 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 (7.20.4.3), the **fgetc** function (7.19.7.1), the **fopen** function (7.19.5.3), the **fputc** function (7.19.7.3), the **setbuf** function (7.19.5.5), the **setvbuf** function (7.19.5.6), the **fgetwc** function (7.24.3.1), the **fputwc** function (7.24.3.3), conversion state (7.24.6), the **mbrtowc** function (7.24.6.3.2), the **wcrtomb** function (7.24.6.3.3).

7.19.4 Operations on files

7.19.4.1 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.

7.19.4.2 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,[210] in which case if the file existed previously it is still known by its original name.

7.19.4.3 The `tmpfile` function

Synopsis

1
```
#include <stdio.h>
FILE *tmpfile(void);
```

Description

2 The **tmpfile** function creates a temporary binary file 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.

Returns

3 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 (7.19.5.3).

210) 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.

7.19.4.4 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.[211)]

3 The **tmpnam** function generates a different string each time it is called, up to **TMP_MAX** times. If it is called more than **TMP_MAX** times, the behavior is implementation-defined.

4 The implementation shall behave as if no library function calls the **tmpnam** function.

Returns

5 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.

7.19.5 File access functions

7.19.5.1 The fclose function

Synopsis

1
```
#include <stdio.h>
int fclose(FILE *stream);
```

Description

2 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. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated.

Returns

211) 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.

3 The **fclose** function returns zero if the stream was successfully closed, or **EOF** if any errors were detected.

7.19.5.2 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 (7.19.5.3).

7.19.5.3 The fopen function

Synopsis

1
```
#include <stdio.h>
FILE *fopen(const char * filename,
     const char * 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.[212)]

212) 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 7.19.2).

`r`	open text file for reading
`w`	truncate to zero length or 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
`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
`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
`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 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.

6 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.

7 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

8 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 (7.19.9).

7.19.5.4 The **freopen** function

Synopsis

1
```
#include <stdio.h>
FILE *freopen(const char * filename,
     const char * 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.[213)]

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**.

7.19.5.5 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**.

Returns

3 The **setbuf** function returns no value.

213) 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.

Forward references: the **setvbuf** function (7.19.5.6).

7.19.5.6 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[214] 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.

7.19.6 Formatted input/output functions

1 The formatted input/output functions[215] shall behave as if there is a sequence point after the actions associated with each specifier.

---

214) 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.

215) The **printf** functions perform writes to memory for the **%n** specifier.

7.19.6.1 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 decimal integer.[216)]

— 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 characters to be written from a string in **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.

216) Note that **0** is taken as a flag, not as the beginning of a field width.

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.)[217)]

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 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; no space padding is performed. 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.

217) The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

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 (finite) 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.[218]

e,E A **double** argument representing a (finite) 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 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 (finite) floating-point number is converted in style **f** or **e** (or in style **F** or **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style **e** (or **E**) is used only if the exponent resulting from such a conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the **#** flag is specified; a decimal-point character appears only if it is followed by a digit.

A **double** argument representing an infinity or NaN is converted in the style

218) When applied to infinite and NaN values, the **-**, **+**, and *space* flag characters have their usual meaning; the **#** and **0** flag characters have no effect.

of an **f** or **F** conversion specifier.

a,A A **double** argument representing a (finite) 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[219] 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[220] 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.[221] 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 characters 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**

---

219) 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.

220) 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.

221) No special provisions are made for multibyte characters.

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 characters (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 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.[222)]

p The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printable 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.[223)] If any argument is not the correct type for the corresponding coversion 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 If **FLT_RADIX** is not a power of 2, 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.[224)] 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

222) Redundant shift sequences may result if multibyte characters have a state-dependent encoding.

223) See ‘‘future library directions’’ (7.26.9).

adjacent decimal strings $L < U$, both having **DECIMAL_DIG** significant digits; the value 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 multibyte members of the extended character set each consist of 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", wstr[5]);
```

will print the following seven lines:

224) 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.

```
|1234567890123|
|   □X□Yabc□Z□W|
|□X□Yabc□Z     |
|     □X□Yabc□Z|
|   □X□Yabc□Z□W|
|       abc□Z□W|
|            □Z|
```

Forward references: conversion state (7.24.6), the **`wcrtomb`** function (7.24.6.3.3).

7.19.6.2 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 nonzero decimal integer 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. 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.

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.

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.[225)]

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. 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**.

225) These white-space characters are not counted against a specified field width.

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).[226)]

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.[226)]

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*).[226)]

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.

The conversion specifier includes all subsequent characters in the **format** string, up to and including the matching right bracket (**]**). The characters

226) 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 still determined on a byte-by-byte basis. The resulting field is nevertheless a sequence of multibyte characters that begins in the initial shift state.

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 interpretation of the input item is implementation-defined. 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.[227)]

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 If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (other than **%n**, if any) is terminated with an input failure.

16 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.

227) See “future library directions” (7.26.9).

17 If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream.[228)]

Returns

18 The **fscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

19 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**.

20 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**.

21 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));
```

22 If the **stdin** stream contains the following lines:

---

228) **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**.

```
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;
```

23 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**.

24 EXAMPLE 5 In these examples, multibyte characters do have a state-dependent encoding, and multibyte members of the extended character set consist of 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.

25 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.

26 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.

27 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.

28 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 (7.20.1.3), the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.20.1.4), conversion state (7.24.6), the **wcrtomb** function (7.24.6.3.3).

7.19.6.3 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.

7.19.6.4 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 any conversion. 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.

7.19.6.5 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**.

7.19.6.6 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.

7.19.6.7 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 any conversion. 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.

7.19.6.8 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.[229)]

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);
}
```

229) 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.

7.19.6.9 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.[229)]

Returns

3 The **vfscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

7.19.6.10 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.[229)]

Returns

3 The **vprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

7.19.6.11 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.[229)]

Returns

3 The **vscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

7.19.6.12 The `vsnprintf` function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(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.[229)] 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**.

7.19.6.13 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.[229)] 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.

7.19.6.14 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.[229)]

Returns

3 The **vsscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

7.19.7 Character input/output functions

7.19.7.1 The **fgetc** function

Synopsis

1
```
#include <stdio.h>
int fgetc(FILE *stream);
```

Description

2 If a next character is present from the input stream pointed to by **stream**, 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 The **fgetc** 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 **fgetc** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **fgetc** returns **EOF**.[230)]

7.19.7.2 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.

230) An end-of-file and a read error can be distinguished by use of the **feof** and **ferror** functions.

7.19.7.3 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**.

7.19.7.4 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.

7.19.7.5 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**.

7.19.7.6 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**.

7.19.7.7 The `gets` function

Synopsis

1
```
#include <stdio.h>
char *gets(char *s);
```

Description

2 The **gets** function reads characters from the input stream pointed to by **stdin**, into the array pointed to by **s**, until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

Returns

3 The **gets** 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.

7.19.7.8 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**.

7.19.7.9 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**.

7.19.7.10 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.

7.19.7.11 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.[231)]

Returns

6 The **ungetc** function returns the character pushed back after conversion, or **EOF** if the operation fails.

Forward references: file positioning functions (7.19.9).

7.19.8 Direct input/output functions

7.19.8.1 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**. 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.

231) See "future library directions" (7.26.9).

7.19.8.2 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**. 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.

7.19.9 File positioning functions

7.19.9.1 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 (7.19.9.3).

7.19.9.2 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 (7.19.9.4).

7.19.9.3 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**.

7.19.9.4 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**.

7.19.9.5 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.

7.19.10 Error-handling functions

7.19.10.1 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.

7.19.10.2 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`.

7.19.10.3 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`.

7.19.10.4 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 (7.21.6.2).

7.20 General utilities **<stdlib.h>**

1 The header **<stdlib.h>** declares five types and several functions of general utility, and defines several macros.[232)]

2 The types declared are **size_t** and **wchar_t** (both described in 7.17),

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 7.17);

EXIT_FAILURE

and

EXIT_SUCCESS

which expand to integer expressions that may 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, the value of which is the maximum value returned by the **rand** function; and

MB_CUR_MAX

which expands to a positive integer expression whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category **LC_CTYPE**), and whose value is never greater than **MB_LEN_MAX**.

7.20.1 String 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.

232) See “future library directions” (7.26.10).

7.20.1.1 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 (7.20.1.3).

7.20.1.2 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 (7.20.1.4).

7.20.1.3 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, where either the decimal-point character or the binary-exponent part is present;

— one of **INF** or **INFINITY**, ignoring case

— one of **NAN** or **NAN(**$n\text{-}char\text{-}sequence_{opt}$**)**, 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, a binary-exponent part, nor a decimal-point character appears, a decimal point is assumed to follow the last digit in the string. 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 sequences is implementation-defined.[233] If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.[234] 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, then 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 and **FLT_RADIX** is not a power of 2, then 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, then the value resulting from the conversion 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 of conversion 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.[235]

Returns

10 The 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, 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.

233) An implementation may use the *n-char-sequence* to determine extra information to be represented in the NaN's significand.

234) The functions honor the sign of zero if floating-point arithmetic supports signed zeros.

235) **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.

7.20.1.4 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 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`. *

7.20.2 Pseudo-random sequence generation functions

7.20.2.1 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`.

3 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.

7.20.2.2 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 implementation shall behave as if no library function calls the **srand** function.

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;
}
```

7.20.3 Memory management functions

1 The order and contiguity of storage allocated by successive calls to the **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 and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly freed or reallocated). 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. The value of a pointer that refers to freed space is indeterminate.

7.20.3.1 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.[236)]

Returns

3 The **calloc** function returns either a null pointer or a pointer to the allocated space.

7.20.3.2 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 the **calloc**, **malloc**, or **realloc** 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.

---

236) Note that this need not be the same as the representation of floating-point zero or a null pointer constant.

7.20.3.3 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.

7.20.3.4 The **realloc** function

Synopsis

1
```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

Description

2 The **realloc** function changes the size of the object pointed to by **ptr** to the size specified by **size**. The contents of the object shall be unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate. 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 the **calloc**, **malloc**, or **realloc** function, or if the space has been deallocated by a call to the **free** or **realloc** function, the behavior is undefined. If the space cannot be allocated, the object pointed to by **ptr** is unchanged. If the **realloc** function returns a null pointer when **size** is zero and **ptr** is not a null pointer, the object it pointed to has been freed.

Returns

3 The **realloc** function returns either a null pointer or a pointer to the possibly moved allocated space. If the object has moved, **ptr** is a pointer that refers to freed space.

7.20.4 Communication with the environment

7.20.4.1 The `abort` function

Synopsis

1
```
#include <stdlib.h>
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 output streams are flushed or open streams closed or temporary files 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.

7.20.4.2 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.

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 **exit** function (7.20.4.3).

7.20.4.3 The `exit` function

Synopsis

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

Description

2 The **exit** function causes normal program termination to occur. If more than one call to the **exit** function is executed by a program, the behavior is undefined.

3 First, all functions registered by the **atexit** function are called, in the reverse order of their registration.[237)]

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.

7.20.4.4 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.

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.

237) Each function is called as many times as it was registered.

7.20.4.5 The **system** function

Synopsis

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

Description

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

Returns

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

7.20.5 Searching and sorting utilities

1 These utilities make use of a comparison function.

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.[238] 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.

238) That is, if the value passed is **p**, then the following expressions are always non-zero:

```
((char *)p - (char *)base) % size == 0
(char *)p >= (char *)base
(char *)p < (char *)base + nmemb * size
```

7.20.5.1 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.[239)]

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.

7.20.5.2 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.

4 If two elements compare as equal, their order in the resulting sorted array is unspecified.

---

239) In practice, the entire array is sorted according to the comparison function.

Returns

5 The `qsort` function returns no value.

7.20.6 Integer arithmetic functions

7.20.6.1 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.[240)]

Returns

3 The `abs`, `labs`, and `llabs`, functions return the absolute value.

7.20.6.2 The `div`, `ldiv`, and `lldiv` functions

Synopsis

1
```
#include <stdlib.h>
div_t div(int numer, int denom);
ldiv_t div(long int numer, long int denom);
lldiv_t div(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 have the same type as the arguments `numer` and `denom`. If either part of the result cannot be represented, the behavior is undefined.

240) The absolute value of the most negative number cannot be represented in two's complement.

7.20.7 Multibyte character 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 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 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.[241] Changing the **LC_CTYPE** category causes the shift state of these functions to be indeterminate.

7.20.7.1 The mblen function

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 shift state of the **mbtowc** function is not affected, it is equivalent to

```
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 (7.20.7.2).

241) 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.

7.20.7.2 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 determines the number of bytes that are contained in the multibyte character pointed to by **s**. It then determines the code for the value of type **wchar_t** that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero.) If the multibyte character is valid and **pwc** is not a null pointer, the **mbtowc** function stores the code in the object pointed to by **pwc**. At most **n** bytes of the array pointed to by **s** will be examined.

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

Returns

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).

4 In no case will the value returned be greater than **n** or the value of the **MB_CUR_MAX** macro.

7.20.7.3 The wctomb function

Synopsis

1
```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Description

2 The **wctomb** function determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is **wchar** (including any change in shift state). It stores the multibyte character representation in the array object pointed to by **s** (if **s** is not a null pointer). At most **MB_CUR_MAX** characters are stored. If the value of **wchar** is zero, the **wctomb** function is left in the initial shift 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 **wchar** 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 **wchar**.

5 In no case will the value returned be greater than the value of the **MB_CUR_MAX** macro.

7.20.8 Multibyte string functions

1 The behavior of the multibyte string functions is affected by the **LC_CTYPE** category of the current locale.

7.20.8.1 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 codes and stores not more than **n** codes into the array pointed to by **pwcs**. No multibyte characters that follow a null character (which is converted into a code with value zero) will be examined or converted. Each multibyte character is converted as if by a call to the **mbtowc** function, except that the shift 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 zero code, if any.[242)]

242) The array will not be null- or zero-terminated if the value returned is **n**.

7.20.8.2 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 codes that correspond to multibyte characters from the array pointed to by **pwcs** into a sequence of 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 code is converted as if by a call to the **wctomb** function, except that the shift 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 code 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.[242)]

7.21 String handling `<string.h>`

7.21.1 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.[243] The type is **`size_t`** and the macro is **`NULL`** (both described in 7.17). 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.

7.21.2 Copying functions

7.21.2.1 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`**.

243) See "future library directions" (7.26.11).

7.21.2.2 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**.

7.21.2.3 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**.

7.21.2.4 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**.[244] If copying takes place between objects that overlap, the behavior is undefined.

244) 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.

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**.

7.21.3 Concatenation functions

7.21.3.1 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**.

7.21.3.2 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.[245] If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **strncat** function returns the value of **s1**.

245) Thus, the maximum number of characters that can end up in the array pointed to by **s1** is **strlen(s1)+n+1**.

Forward references: the `strlen` function (7.21.6.3).

7.21.4 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.

7.21.4.1 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`.[246]

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`.

7.21.4.2 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 pointed to by `s2`.

---

246) 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.

7.21.4.3 The `strcoll` function

Synopsis

1
```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Description

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

2 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.

7.21.4.4 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**.

7.21.4.5 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)
```

7.21.5 Search functions

7.21.5.1 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**.

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.

7.21.5.2 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.

7.21.5.3 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.

7.21.5.4 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**.

7.21.5.5 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.

7.21.5.6 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.

7.21.5.7 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`.

7.21.5.8 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 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 1

```
#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
```

7.21.6 Miscellaneous functions

7.21.6.1 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**.

7.21.6.2 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 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.

7.21.6.3 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.

7.22 Type-generic math **<tgmath.h>**

1 The header **<tgmath.h>** includes the headers **<math.h>** and **<complex.h>** and defines several *type-generic macros*.

7.22.1 Type-generic macros

1 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*.[247] 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.[248]

2 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**.

3 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**.

247) Like other function-like macros in Standard libraries, each type-generic macro can be suppressed to make available the corresponding ordinary function.

248) If the type of the argument is incompatible 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.

4 For each unsuffixed function in **<math.h>** without a **c**-prefixed counterpart in **<complex.h>**, 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**	**nextafterx**	

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.

5 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.

6 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
`nextafterx(f, ld)`	`nextafterxf(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)`
`cabs(fc)`	`cabsf(fc)`
`carg(dc)`	`carg(dc)`, the function
`cproj(ldc)`	`cprojl(ldc)`

7.23 Date and time `<time.h>`

7.23.1 Components of time

1 The header **`<time.h>`** defines four 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 7.17);

`CLOCKS_PER_SEC`

which expands to a constant expression with the type **`clock_t`** described below, and which is the number per second of the value returned by the **`clock`** function;

`_NO_LEAP_SECONDS`

which expands to an integral constant expression of type **`int`** with a value outside the range [−3600, +3600] (described in 7.23.2.4). and

`_LOCALTIME`

which expands to an integral constant expression of type **`int`** with a value outside the range [−14400, +14400] (described in 7.23.2.4).

3 The types declared are **`size_t`** (described in 7.17);

`clock_t`

and

`time_t`

which are arithmetic types capable of representing times; and

`struct tm`

and

`struct tmx`

which hold the components of a calendar time, called the *broken-down time*.

4 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.[249)]

249) The range [0, 60] for **`tm_sec`** allows for a positive leap second.

```
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.

5 The **tmx** structure shall contain all the members of **struct tm** in a manner such that all these members are part of a common initial subsequence. In addition, it contains the members:

```
int tm_version;     // version number
int tm_zone;        // time zone offset in minutes
                    // from UTC [−1439, +1439]
int tm_leapsecs;    // number of leap seconds applied
void *tm_ext;       // extension block
size_t tm_extlen;   // size of the extension block
```

The meaning of **tm_isdst** is also different: it is the positive number of minutes of offset if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and −1 if the information is not available. A positive value for **tm_zone** indicates a time that is ahead of Coordinated Universal Time (UTC). The implementation or a future version of this International Standard may include further members in a separate object. If so, the **tm_ext** member shall point to this object and the **tm_extlen** object shall be its size. Otherwise, the **tm_ext** member shall be a null pointer and the value of the **tm_extlen** object is unspecified.

7.23.2 Time manipulation functions

7.23.2.1 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`.[250]

7.23.2.2 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`.

250) 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.

7.23.2.3 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.[251] 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.

3 The normalization process shall be as described in 7.23.2.6.

4 If the call is successful, a second call to the **mktime** function with the resulting **struct tm** value shall always leave it unchanged and return the same value as the first call. Furthermore, if the normalized time is exactly representable as a **time_t** value, then the normalized broken-down time and the broken-down time generated by converting the result of the **mktime** function by a call to **localtime** shall be identical.

Returns

5 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**.

6 EXAMPLE What day of the week is July 4, 2001?

```
#include <stdio.h>
#include <time.h>
static const char *const wday[] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday", "-unknown-"
};
struct tm time_str;
/* ... */
```

251) 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.

```
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]);
```

7.23.2.4 The **mkxtime** function

Synopsis

1
```
#include <time.h>
time_t mkxtime(struct tmx *timeptr);
```

Description

2 The **mkxtime** function has the same behavior and result as the **mktime** function except that it takes into account the values of the additional members of **struct tmx**.

3 If the value of the **tm_version** member is not 1, the behavior is undefined. If the implementation cannot determine the relationship between local time and UTC, it shall set the **tm_zone** member of the pointed-to structure to **_LOCALTIME**. Otherwise, if the **tm_zone** member was **_LOCALTIME**, it shall be set to the offset of local time from UTC, including the effects of the value of the **tm_isdst** member; otherwise, the original value of the **tm_isdst** member does not affect the result.

4 If the **tm_leapsecs** member is equal to **_NO_LEAP_SECONDS**, then the implementation shall determine the number of leap seconds that apply and set the member accordingly (or use 0 if it cannot determine it); otherwise, it shall use the number of leap seconds given. The **tm_leapsecs** member shall then be set to the number of leap seconds actually applied to produce the value represented by the structure, or to **_NO_LEAP_SECONDS** if it was not possible to determine it.

5 If the call is successful, a second call to the **mkxtime** function with the resulting **struct tmx** value shall always leave it unchanged and return the same value as the first call. Furthermore, if the normalized time is exactly representable as a **time_t** value, then the normalized broken-down time and the broken-down time generated by converting the result of the **mkxtime** function by a call to **zonetime** (with **zone** set to the value of the **tm_zone** member) shall be identical.

Returns

6 The **mkxtime** 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**.

7.23.2.5 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.

7.23.2.6 Normalization of broken-down times

1 A broken-down time is normalized by the `mkxtime` function in the following manner. A broken-down time is normalized by the `mktime` function in the same manner, but as if the `struct tm` structure had been replaced by a `struct tmx` structure containing the same values except:

`tm_version` is 1

`tm_zone` is `_LOCALTIME`

`tm_leapsecs` is `_NO_LEAP_SECONDS`

`tm_isdst` is −1, 0, or an implementation-defined positive value according to whether the original member is less than, equal to, or greater than zero

2 If any of the following members is outside the indicated range (where L is `LONG_MAX`/8), the behavior is undefined:

`tm_year` $[-L/366, +L/366]$

`tm_mon` $[-L/31, +L/31]$

`tm_mday` $[-L, +L]$

`tm_hour` $[-L/3600, +L/3600]$

`tm_min` $[-L/60, +L/60]$

`tm_sec` $[-L, +L]$

`tm_leapsecs` $[-L, +L]$ or `_NO_LEAP_SECONDS`

`tm_zone` $[-L/60, +L/60]$

`tm_isdst` $[-L/60, +L/60]$ or `_LOCALTIME`

The `tm_version` member shall be 1.

3 Values **S** and **D** shall be determined as follows:

```
#define QUOT(a,b) ((a)>0 ? (a)/(b) : -(((b)-(a)-1)/(b)))
#define REM(a,b) ((a)-(b)*QUOT(a,b))

SS = tm_hour*3600 + tm_min*60 + tm_sec +
     (tm_leapsecs == _NO_LEAP_SECONDS ? X1 :
          tm_leapsecs) -
     (tm_zone == _LOCALTIME ? X2 : tm_zone) * 60;

// X1 is the appropriate number of leap seconds, determined by
// the implementation, or 0 if it cannot be determined.
// X2 is the appropriate offset from local time to UTC,
// determined by the implementation, or
// (tm_isdst >= 0 ? tm_isdst : 0)
// if the offset cannot be determined

M = REM(tm_mon, 12);
Y = tm_year + 1900 + QUOT(tm_mon, 12);
Z = Y - (M < 2 ? 1 : 0);
D = Y*365 + (Z/400)*97 + (Z%400)/4 +
     M[(int []){0,31,59,90,120,151,181,212,243,273,
          304,335}] +
     tm_mday + QUOT(SS, 86400);
S = REM(SS, 86400);
```

4 The normalized broken-down time shall produce the same values of **S** and **D** (though possibly different values of **M**, **Y**, and **Z)** as the original broken-down time.[252)]

7.23.3 Time conversion functions

1 Except for the **strftime** and **strfxtime** functions, 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. The implementation shall behave as if no other library functions call these functions.

252) The effect of the above rules is to consistently use the Gregorian calendar, regardless of which calendar was in use in which year. In particular, the years 1100 and −300 are not leap years, while the years 1200 and −400 are (these 4 years correspond to **tm_year** values of −800, −2200, −700, and −2300 respectively, and the last of these is 401 B.C.E.). In the normalized broken-down time, **tm_wday** is equal to **QUOT(D-2,7)**.

7.23.3.1 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 result[26];

     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;
}
```

Returns

3 The **asctime** function returns a pointer to the string.

7.23.3.2 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 (7.23.3.4).

7.23.3.3 The `gmtime` function

Synopsis

1
```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

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.

7.23.3.4 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.

7.23.3.5 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 7.23.1]
- `%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 7.23.1]
- **%X** is replaced by the locale's appropriate time representation. [all specified in 7.23.1]
- **%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 the **E** or **O** modifier characters 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.
%Ou 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 **1**.

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 %d %T %Y**".
%p one of "**am**" or "**pm**".

%r equivalent to "**%I:%M:%S %p**".
%x equivalent to "**%A %B %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.

7.23.3.6 The strfxtime function

Synopsis

1
```
#include <time.h>
size_t strfxtime(char * restrict s,
     size_t maxsize,
     const char * restrict format,
     const struct tmx * restrict timeptr);
```

Description

2 The behavior and result of the **strfxtime** is identical to that of the **strftime** function, except that the **timeptr** parameter has a different type, and the **%z** and **%Z** conversion specifiers depend on **tm_zone** in addition to **tm_isdst**.

7.23.3.7 The zonetime function

Synopsis

1
```
#include <time.h>
struct tmx *zonetime(const time_t *timer, int zone);
```

Description

2 The **zonetime** function converts the calendar time pointed to by **timer** into a broken-down time as represented in the specified time zone. The **tm_version** member is set to 1. If the implementation cannot determine the relationship between local time and UTC, it shall set the **tm_zone** member to **_LOCALTIME**; otherwise, it shall set the **tm_zone** member to the value of **zone** unless the latter is **_LOCALTIME**, in which case it shall set it to the offset of local time from UTC. The value shall include the effect of Daylight Saving Time, if in effect. The **tm_leapsecs** member shall be set to the number of leap seconds (the UTC-UT1 offset) applied in the result[253] if it can be determined, or to the value **_NO_LEAP_SECONDS** if it cannot (and so none were applied).

253) If the **tm_sec** member is set to 60, that leap second shall not be included in the value of **tm_leapsecs**.

Returns

3 The **`zonetime`** function returns a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to the specified time zone.

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

7.24.1 Introduction

1 The header **`<wchar.h>`** declares four data types, one tag, four macros, and many functions.[254)]

2 The types declared are **`wchar_t`** and **`size_t`** (both described in 7.17);

`mbstate_t`

which is an 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`

described in 7.25.1; and

`struct tm`

and

`struct tmx`

which are declared as incomplete structure types, the contents of which are described in 7.23.1.

3 The macros defined are **`NULL`** (described in 7.17);

`WCHAR_MAX`

which is the maximum value representable by an object of type **`wchar_t`**;[255)]

`WCHAR_MIN`

which is the minimum value representable by an object of type **`wchar_t`**; and

`WEOF`

described in 7.25.1.

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;

---

254) See ‘‘future library directions’’ (7.26.12).

255) The values **`WCHAR_MAX`** and **`WCHAR_MIN`** do not necessarily correspond to members of the extended character set.

— Functions for wide-string date and time conversion; and

— Functions that provide extended capabilities for conversion between multibyte and wide-character sequences.

5 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.

7.24.2 Formatted wide-character input/output functions

1 The formatted wide-character input/output functions[256] shall behave as if there is a sequence point after the actions associated with each specifier.

7.24.2.1 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.

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 decimal integer.[257]

256) The **`fwprintf`** functions perform writes to memory for the **`%n`** specifier.

257) Note that **`0`** is taken as a flag, not as the beginning of a field width.

— 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 from a string in **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 specified.)[258)]

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 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; no space padding is performed. If the **0** and **-** flags both appear, the **0** flag is

258) The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

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 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 (finite) 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.[259)]

e,E A **double** argument representing a (finite) 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.

---

259) 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.

g,G A **double** argument representing a (finite) floating-point number is converted in style **f** or **e** (or in style **F** or **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style **e** (or **E**) is used only if the exponent resulting from such a conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the **#** flag is specified; a decimal-point wide character appears only if it is followed by a digit.

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 (finite) 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[260] 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[261] 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.

---

260) 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.

261) 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.

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 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 printable 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.[262] If any argument is not the correct type for the corresponding coversion 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 If **FLT_RADIX** is not a power of 2, 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.

262) See ‘‘future library directions’’ (7.26.12).

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.[263] 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 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 (7.24.6.1.1), the **mbrtowc** function (7.24.6.3.2).

263) 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.

7.24.2.2 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 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 nonzero decimal integer 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. 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.

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.

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.[264)]

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 sequence. 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

264) These white-space wide characters are not counted against a specified field width.

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 **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 interpretation of the input item is implementation-defined. 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.[265)]

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 If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any wide characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (other than **%n**, if any) is terminated with an input failure.

16 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.

17 If conversion terminates on a conflicting input wide character, the offending input wide character is left unread in the input stream.[266)]

Returns

18 The **fwscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

19 EXAMPLE 1 The call:

265) See “future library directions” (7.26.12).

266) **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**.

```
#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);
```

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`**.

20 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 (7.24.4.1.1), the **`wcstol`**, **`wcstoll`**, **`wcstoul`**, and **`wcstoull`** functions (7.24.4.1.2), the **`wcrtomb`** function (7.24.6.3.3).

7.24.2.3 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.

7.24.2.4 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 any conversion. 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.

7.24.2.5 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.[267)]

Returns

3 The **vfwprintf** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred. |

4 EXAMPLE The following shows the use of the **vfwprintf** function in a general error-reporting routine.

---

267) As the functions **vfwprintf**, **vswprintf**, **vfwscanf**, **vwprintf**, **vwscanf**, and **vswscanf** invoke the **va_arg** macro, the value of **arg** after the return is indeterminate.

```
#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);
}
```

7.24.2.6 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.[267)]

Returns

3 The **vfwscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

7.24.2.7 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.[267)]

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.

7.24.2.8 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.[267)]

Returns

3 The **vswscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

7.24.2.9 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.[267)]

Returns

3 The **vwprintf** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

7.24.2.10 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.[267)]

Returns

3 The **vwscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. 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.

7.24.2.11 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.

7.24.2.12 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 any conversion. 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.

7.24.3 Wide-character input/output functions

7.24.3.1 The `fgetwc` function

Synopsis

1

```
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *stream);
```

Description

2 If a next wide character is present from the input stream pointed to by `stream`, the `fgetwc` function obtains that wide character and advances the associated file position indicator for the stream (if defined).

Returns

3 The `fgetwc` function returns the next wide 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 `fgetwc` returns `WEOF`. If a read error occurs, the error indicator for the stream is set and `fgetwc` returns `WEOF`. If an encoding error occurs (including too few bytes), the value of the macro `EILSEQ` is stored in `errno` and `fgetwc` returns `WEOF`.[268)]

7.24.3.2 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.

---

268) 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.

7.24.3.3 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 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`.

7.24.3.4 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.

7.24.3.5 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.[269] 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.

7.24.3.6 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**.

269) If the orientation of the stream has already been determined, **fwide** does not change it.

7.24.3.7 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**.

7.24.3.8 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**.

7.24.3.9 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**.

7.24.3.10 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.

7.24.4 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.

7.24.4.1 Wide-string numeric conversion functions

7.24.4.1.1 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, where either the decimal-point wide character or the binary-exponent part is present;

— one of **INF** or **INFINITY**, or any other wide string equivalent except for case

— one of **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, a binary-exponent part, nor a decimal-point wide character appears, a decimal point is assumed to follow the last digit in the wide string. 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 sequences is implementation-defined.[270] If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.[271] A pointer to the 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, then 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 and **FLT_RADIX** is not a power of 2, then 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, then the value resulting from the conversion 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 of conversion 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.[272]

Returns

10 The 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, plus or minus **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** is returned (according to the return

270) An implementation may use the n-wchar sequence to determine extra information to be represented in the NaN's significand.

271) The functions honor the sign of zero if floating-point arithmetic supports signed zeros.

272) **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.

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.

7.24.4.1.2 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**.

7.24.4.2 Wide-string copying functions

7.24.4.2.1 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**.

7.24.4.2.2 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**.[273]

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**.

7.24.4.3 Wide-string concatenation functions

7.24.4.3.1 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**.

273) 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.

7.24.4.3.2 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.[274)]

Returns

3 The **wcsncat** function returns the value of **s1**.

7.24.4.4 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**.

7.24.4.4.1 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**.

274) Thus, the maximum number of wide characters that can end up in the array pointed to by **s1** is **wcslen(s1)+n+1**.

7.24.4.4.2 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 wide string pointed to by **s2** when both are interpreted as appropriate to the current locale.

7.24.4.4.3 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**.

7.24.4.4.4 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)
```

7.24.4.5 Wide-string search functions

7.24.4.5.1 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.

7.24.4.5.2 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.

7.24.4.5.3 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.

7.24.4.5.4 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`**.

7.24.4.5.5 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.

7.24.4.5.6 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.

7.24.4.5.7 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**.

7.24.4.5.8 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 points to the token L"a"
t = wcstok(str1, L"?", &ptr1);

// t points to the token L"??b"
t = wcstok(NULL, L",", &ptr1);

// t is a null pointer
t = wcstok(str2, L" \t", &ptr2);

// t points to the token L"c"
t = wcstok(NULL, L"#,", &ptr1);

// t is a null pointer
t = wcstok(NULL, L"?", &ptr1);
```

7.24.4.6 Wide-character array functions

1 These functions operate on arrays of type **wchar_t** whose size is specified by a separate count argument. These functions are not affected by locale, and all **wchar_t** values are treated identically. The null wide character and **wchar_t** values not corresponding to valid multibyte characters are not treated specially.

2 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**.

3 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 stated explicitly 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.

7.24.4.6.1 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.

7.24.4.6.2 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**.

7.24.4.6.3 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**.

7.24.4.6.4 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**.

7.24.4.6.5 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**.

7.24.5 Wide-character time conversion functions

7.24.5.1 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.

7.24.5.2 The `wcsfxtime` function

Synopsis

1
```
#include <time.h>
#include <wchar.h>
size_t wcsfxtime(wchar_t * restrict s,
     size_t maxsize,
     const wchar_t * restrict format,
     const struct tmx * restrict timeptr);
```

Description

2 The **wcsfxtime** function is equivalent to the **wcsftime** function, except that the **timeptr** parameter has a different type, and the **%z** and **%Z** conversion specifiers depend on both the **tm_zone** and **tm_isdst** members.

7.24.6 Extended multibyte and 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", 7.24.6.3 and 7.24.6.4 — 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.[275)]

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 pointed-to object is altered as needed to track the shift state, and the position within a multibyte character, for the associated multibyte character sequence.

7.24.6.1 Single-byte wide-character conversion functions

7.24.6.1.1 The btowc function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t btowc(int c);
```

Description

2 The **btowc** function determines whether **c** constitutes a valid (one-byte) multibyte character in the initial shift state.

Returns

3 The **btowc** returns **WEOF** if **c** has the value **EOF** or if **(unsigned char)c** does not constitute a valid (one-byte) multibyte character in the initial shift state. Otherwise, it returns the wide-character representation of that character.

275) 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.

7.24.6.1.2 The `wctob` function

Synopsis

1
```
#include <stdio.h>
#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** 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**.

7.24.6.2 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 pointed-to **mbstate_t** object describes an initial conversion state.

Returns

3 The **mbsinit** function returns nonzero if **ps** is a null pointer or if the pointed-to object describes an initial conversion state; otherwise, it returns zero.

7.24.6.3 Restartable multibyte/wide-character conversion functions

1 These functions differ from the corresponding multibyte character functions of 7.20.7 (**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 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.

7.24.6.3.1 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 (7.24.6.3.2).

7.24.6.3.2 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 completed, 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).

positive 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).[276)]

(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 undefined.

7.24.6.3.3 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 resulting bytes 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 undefined.

276) 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).

7.24.6.4 Restartable multibyte/wide-string conversion functions

1 These functions differ from the corresponding multibyte string functions of 7.20.8 (**`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 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.

7.24.6.4.1 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, beginning 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`** codes have been stored into the array pointed to by **`dst`**.[277] 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 undefined.

---

277) Thus, the value of **`len`** is ignored if **`dst`** is a null pointer.

Otherwise, it returns the number of multibyte characters successfully converted, not including the terminating null (if any).

7.24.6.4.2 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, beginning 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 code 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.[278)]

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 code 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 undefined. Otherwise, it returns the number of bytes in the resulting multibyte character sequence, not including the terminating null (if any).

278) 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.

7.25 Wide-character classification and mapping utilities **<wctype.h>**

7.25.1 Introduction

1 The header **<wctype.h>** declares three data types, one macro, and many functions.[279)]

2 The types declared are

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);[280)]

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

which expands to a constant expression of type **wint_t** whose value does not correspond to any member of the extended character set.[281)] 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 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.

279) See ‘‘future library directions’’ (7.26.13).

280) **wchar_t** and **wint_t** can be the same integer type.

281) The value of the macro **WEOF** may differ from that of **EOF** and need not be negative.

6 The behavior of these functions is affected by the **LC_CTYPE** category of the current locale.

7.25.2 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.

7.25.2.1 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 Except for the **iswgraph** and **iswpunct** functions with respect to printing, white-space, wide characters other than **L' '**, each of the following functions returns true for each wide character that corresponds (as if by a call to the **wctob** function) to a character (byte) for which the corresponding character testing function from 7.4.1 returns true.[282)]

Forward references: the **wctob** function (7.24.6.1.2).

7.25.2.1.1 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.

---

282) 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.

7.25.2.1.2 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 wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.[283)]

7.25.2.1.3 The **iswcntrl** function

Synopsis

1
```
#include <wctype.h>
int iswcntrl(wint_t wc);
```

Description

2 The **iswcntrl** function tests for any control wide character.

7.25.2.1.4 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).

283) The functions **iswlower** and **iswupper** test true or false separately for each of these additional wide characters; all four combinations are possible.

7.25.2.1.5 The **iswgraph** function

Synopsis

1
```
#include <wctype.h>
int iswgraph(wint_t wc);
```

Description

2 The **iswgraph** function tests for any wide character for which **iswprint** is true and **iswspace** is false.[284)]

7.25.2.1.6 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.

7.25.2.1.7 The **iswprint** function

Synopsis

1
```
#include <wctype.h>
int iswprint(wint_t wc);
```

Description

2 The **iswprint** function tests for any printing wide character.

284) 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, basic execution characters other than `' '`.

7.25.2.1.8 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.[284)]

7.25.2.1.9 The **iswspace** function

Synopsis

1
```
#include <wctype.h>
int iswspace(wint_t wc);
```

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.

7.25.2.1.10 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.

7.25.2.1.11 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).

7.25.2.2 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 (7.25.2.1).

7.25.2.2.1 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 (7.25.2.1) in the comment that follows the expression:

```
iswctype(wc, wctype("alnum"))     // iswalnum(wc)
iswctype(wc, wctype("alpha"))     // iswalpha(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**.

7.25.2.2.2 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.

Forward references: the **iswctype** function (7.25.2.2.1).

7.25.3 Wide-character mapping utilities

1 The header **<wctype.h>** declares several functions useful for mapping wide characters.

7.25.3.1 Wide-character case mapping functions

7.25.3.1.1 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.

7.25.3.1.2 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 characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.25.3.2 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 (7.25.3.1).

7.25.3.2.1 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 (7.25.3.1) 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**.

7.25.3.2.2 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.

7.26 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.

7.26.1 Complex arithmetic **<complex.h>**

1 The function names

cerf	**cexpm1**	**clog2**
cerfc	**clog10**	**cgamma**
cexp2	**clog1p**	**clgamma**

and the same names suffixed with **f** or **l** are reserved for the functions with complex arguments and return values.

7.26.2 Character handling **<ctype.h>**

1 Function names that begin with either **is** or **to**, and a lowercase letter (possibly followed by any combination of digits, letters, and underscore) may be added to the declarations in the **<ctype.h>** header.

7.26.3 Errors **<errno.h>**

1 Macros that begin with **E** and a digit or **E** and an uppercase letter (possibly followed by any combination of digits, letters, and underscore) may be added to the declarations in the **<errno.h>** header.

7.26.4 Format conversion of integer types **<inttypes.h>**

1 Macro names beginning with **PRI** or **SCN** followed by any lower case letter or **X** may be added to the macros defined in the **<inttypes.h>** header.

7.26.5 Localization **<locale.h>**

1 Macros that begin with **LC_** and an uppercase letter (possibly followed by any combination of digits, letters, and underscore) may be added to the definitions in the **<locale.h>** header.

7.26.6 Signal handling **<signal.h>**

1 Macros that begin with either **SIG** and an uppercase letter or **SIG_** and an uppercase letter (possibly followed by any combination of digits, letters, and underscore) may be added to the definitions in the **<signal.h>** header.

7.26.7 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.

7.26.8 Integer types <stdint.h>

1 Type 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** or **_MIN**, may be added to the macros defined in the **<stdint.h>** header.

7.26.9 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.

7.26.10 General utilities <stdlib.h>

1 Function names that begin with **str** and a lowercase letter (possibly followed by any combination of digits, letters, and underscore) may be added to the declarations in the **<stdlib.h>** header.

7.26.11 String handling <string.h>

1 Function names that begin with **str**, **mem**, or **wcs** and a lowercase letter (possibly followed by any combination of digits, letters, and underscore) may be added to the declarations in the **<string.h>** header.

7.26.12 Extended multibyte and wide-character utilities <wchar.h>

1 Function names that begin with **wcs** and a lowercase letter (possibly followed by any combination of digits, letters, and underscore) 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.

7.26.13 Wide-character classification and mapping utilities <wctype.h>

1 Function names that begin with **is** or **to** and a lowercase letter (possibly followed by any combination of digits, letters, and underscore) may be added to the declarations in the **<wctype.h>** header.

Annex A
(informative)

Language syntax summary

1 NOTE The notation is described in the introduction to clause 6 (Language).

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 *universal-character-name* that cannot be one of the above
each non-white-space character that cannot be one of the above

A.1.2 Keywords

(6.4.1) *keyword:* one of

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

A.1.3 Identifiers

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

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

(6.4.2) *nondigit:* one of
universal-character-name
_ 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) *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* **'**

(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:*
"*s-char-sequence*$_{opt}$**"**
L"*s-char-sequence*$_{opt}$**"**

(6.4.5) *s-char-sequence:*
s-char
s-char-sequence s-char

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

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* `)`

(6.5.2) *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 , }

(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* **)**

(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$_{opt}$ **;**

(6.7) *declaration-specifiers:*
  storage-class-specifier declaration-specifiers$_{opt}$
  type-specifier declaration-specifiers$_{opt}$
  type-qualifier declaration-specifiers$_{opt}$
  function-specifier declaration-specifiers$_{opt}$

(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
  auto
  register

(6.7.2) *type-specifier:*
  void
  char
  short
  int
  long
  float
  double
  signed
  unsigned
  _Bool
  _Complex
  _Imaginary
  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 **`;`**

(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*

(6.7.3) *type-qualifier:*
 `const`
 `restrict`
 `volatile`

(6.7.4) *function-specifier:*
 `inline`

(6.7.5) *declarator:*
 pointer$_{opt}$ *direct-declarator*

(6.7.5) *direct-declarator:*
identifier
(*declarator* **)**
direct-declarator **[** *assignment-expression*$_{opt}$ **]**
direct-declarator **[*]**
direct-declarator **(** *parameter-type-list* **)**
direct-declarator **(** *identifier-list*$_{opt}$ **)**

(6.7.5) *pointer:*
***** *type-qualifier-list*$_{opt}$
***** *type-qualifier-list*$_{opt}$ *pointer*

(6.7.5) *type-qualifier-list:*
type-qualifier
type-qualifier-list type-qualifier

(6.7.5) *parameter-type-list:*
parameter-list
parameter-list **, ...**

(6.7.5) *parameter-list:*
parameter-declaration
parameter-list **,** *parameter-declaration*

(6.7.5) *parameter-declaration:*
declaration-specifiers declarator
declaration-specifiers abstract-declarator$_{opt}$

(6.7.5) *identifier-list:*
identifier
identifier-list **,** *identifier*

(6.7.6) *type-name:*
specifier-qualifier-list abstract-declarator$_{opt}$

(6.7.6) *abstract-declarator:*
pointer
pointer$_{opt}$ *direct-abstract-declarator*

(6.7.6) *direct-abstract-declarator:*
(*abstract-declarator* **)**
direct-abstract-declarator$_{opt}$ **[** *assignment-expression*$_{opt}$ **]**
direct-abstract-declarator **[*]**
direct-abstract-declarator$_{opt}$ **(** *parameter-type-list*$_{opt}$ **)**

(6.7.7) *typedef-name:*
identifier

(6.7.8) *initializer:*
assignment-expression
`{` *initializer-list* `}`
`{` *initializer-list* `,` `}`

(6.7.8) *initializer-list:*
designation$_{opt}$ *initializer*
initializer-list `,` *designation*$_{opt}$ *initializer*

(6.7.8) *designation:*
designator-list `=`

(6.7.8) *designator-list:*
designator
designator-list designator

(6.7.8) *designator:*
`[` *constant-expression* `]`
`.` *identifier*

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:*
$pp\text{-}tokens_{opt}$ *new-line*
if-section
control-line

(6.10.1) *if-section:*
if-group $elif\text{-}groups_{opt}$ $else\text{-}group_{opt}$ *endif-line*

(6.10.1) *if-group:*
  # if *constant-expression new-line group*$_{opt}$
  # ifdef *identifier new-line group*$_{opt}$
  # ifndef *identifier new-line group*$_{opt}$

(6.10.1) *elif-groups:*
  elif-group
  elif-groups elif-group

(6.10.1) *elif-group:*
  # elif *constant-expression new-line group*$_{opt}$

(6.10.1) *else-group:*
  # else *new-line group*$_{opt}$

(6.10.1) *endif-line:*
  # endif *new-line*

control-line:
(6.10.2) **# include** *pp-tokens new-line*
(6.10.3) **# define** *identifier replacement-list new-line*
(6.10.3) **# define** *identifier lparen identifier-list*$_{opt}$ **)**
  replacement-list new-line
(6.10.3) **# define** *identifier lparen* **...)** *replacement-list new-line*
(6.10.3) **# define** *identifier lparen identifier-list* **, ...)**
  replacement-list new-line
(6.10.3) **# undef** *identifier new-line*
(6.10.4) **# line** *pp-tokens new-line*
(6.10.5) **# error** *pp-tokens*$_{opt}$ *new-line*
(6.10.6) **# pragma** *pp-tokens*$_{opt}$ *new-line*
(6.10.7) **#** *new-line*

(6.10.3) *lparen:*
  the left-parenthesis character without preceding white space

(6.10.3) *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
void assert(int expression);
```

B.2 Complex <complex.h>

```
complex            imaginary          I
_Complex_I         _Imaginary_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 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 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
```

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
void feclearexcept(int excepts);
void fegetexceptflag(fexcept_t *flagp,
     int excepts);
void feraiseexcept(int excepts);
void fesetexceptflag(const fexcept_t *flagp, int excepts);
int fetestexcept(int excepts);
int fegetround(void);
int fesetround(int round);
void fegetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
void fesetenv(const fenv_t *envp);
void feupdateenv(const fenv_t *envp);
```

B.6 Characteristics of floating types `<float.h>`

`FLT_ROUNDS`	`DBL_MIN_EXP`	`FLT_MAX`
`FLT_EVAL_METHOD`	`LDBL_MIN_EXP`	`DBL_MAX`
`FLT_RADIX`	`FLT_MIN_10_EXP`	`LDBL_MAX`
`FLT_MANT_DIG`	`DBL_MIN_10_EXP`	`FLT_EPSILON`
`DBL_MANT_DIG`	`LDBL_MIN_10_EXP`	`DBL_EPSILON`
`LDBL_MANT_DIG`	`FLT_MAX_EXP`	`LDBL_EPSILON`
`DECIMAL_DIG`	`DBL_MAX_EXP`	`FLT_MIN`
`FLT_DIG`	`LDBL_MAX_EXP`	`DBL_MIN`
`DBL_DIG`	`FLT_MAX_10_EXP`	`LDBL_MIN`
`LDBL_DIG`	`DBL_MAX_10_EXP`	
`FLT_MIN_EXP`	`LDBL_MAX_10_EXP`	

B.7 Format conversion of integer types `<inttypes.h>`

`PRId8`	`PRIo8`	`PRIx8`	`SCNd8`
`PRId16`	`PRIo16`	`PRIx16`	`SCNd16`
`PRId32`	`PRIo32`	`PRIx32`	`SCNd32`
`PRId64`	`PRIo64`	`PRIx64`	`SCNd64`
`PRIdLEAST8`	`PRIoLEAST8`	`PRIxLEAST8`	`SCNdLEAST8`
`PRIdLEAST16`	`PRIoLEAST16`	`PRIxLEAST16`	`SCNdLEAST16`
`PRIdLEAST32`	`PRIoLEAST32`	`PRIxLEAST32`	`SCNdLEAST32`
`PRIdLEAST64`	`PRIoLEAST64`	`PRIxLEAST64`	`SCNdLEAST64`
`PRIdFAST8`	`PRIoFAST8`	`PRIxFAST8`	`SCNdFAST8`
`PRIdFAST16`	`PRIoFAST16`	`PRIxFAST16`	`SCNdFAST16`
`PRIdFAST32`	`PRIoFAST32`	`PRIxFAST32`	`SCNdFAST32`
`PRIdFAST64`	`PRIoFAST64`	`PRIxFAST64`	`SCNdFAST64`
`PRIdMAX`	`PRIoMAX`	`PRIxMAX`	`SCNdMAX`
`PRIdPTR`	`PRIoPTR`	`PRIxPTR`	`SCNdPTR`
`PRIi8`	`PRIu8`	`PRIX8`	`SCNi8`
`PRIi16`	`PRIu16`	`PRIX16`	`SCNi16`
`PRIi32`	`PRIu32`	`PRIX32`	`SCNi32`
`PRIi64`	`PRIu64`	`PRIX64`	`SCNi64`
`PRIiLEAST8`	`PRIuLEAST8`	`PRIXLEAST8`	`SCNiLEAST8`
`PRIiLEAST16`	`PRIuLEAST16`	`PRIXLEAST16`	`SCNiLEAST16`
`PRIiLEAST32`	`PRIuLEAST32`	`PRIXLEAST32`	`SCNiLEAST32`
`PRIiLEAST64`	`PRIuLEAST64`	`PRIXLEAST64`	`SCNiLEAST64`
`PRIiFAST8`	`PRIuFAST8`	`PRIXFAST8`	`SCNiFAST8`
`PRIiFAST16`	`PRIuFAST16`	`PRIXFAST16`	`SCNiFAST16`
`PRIiFAST32`	`PRIuFAST32`	`PRIXFAST32`	`SCNiFAST32`
`PRIiFAST64`	`PRIuFAST64`	`PRIXFAST64`	`SCNiFAST64`
`PRIiMAX`	`PRIuMAX`	`PRIXMAX`	`SCNiMAX`
`PRIiPTR`	`PRIuPTR`	`PRIXPTR`	`SCNiPTR`

SCNo8	**SCNoFAST64**	**SCNuFAST8**	**SCNxLEAST16**
SCNo16	**SCNoMAX**	**SCNuFAST16**	**SCNxLEAST32**
SCNo32	**SCNoPTR**	**SCNuFAST32**	**SCNxLEAST64**
SCNo64	**SCNu8**	**SCNuFAST64**	**SCNxFAST8**
SCNoLEAST8	**SCNu16**	**SCNuMAX**	**SCNxFAST16**
SCNoLEAST16	**SCNu32**	**SCNuPTR**	**SCNxFAST32**
SCNoLEAST32	**SCNu64**	**SCNx8**	**SCNxFAST64**
SCNoLEAST64	**SCNuLEAST8**	**SCNx16**	**SCNxMAX**
SCNoFAST8	**SCNuLEAST16**	**SCNx32**	**SCNxPTR**
SCNoFAST16	**SCNuLEAST32**	**SCNx64**	
SCNoFAST32	**SCNuLEAST64**	**SCNxLEAST8**	

```
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       INFINITY      FP_SUBNORMAL  FP_ILOGB0       *
double_t      NAN           FP_ZERO       FP_ILOGBNAN
HUGE_VAL      FP_INFINITE   FP_FAST_FMA
HUGE_VALF     FP_NAN        FP_FAST_FMAF
HUGE_VALL     FP_NORMAL     FP_FAST_FMAL
```

```
#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 nextafterx(double x, long double y);
float nextafterxf(float x, long double y);
long double nextafterxl(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);
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 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.15 Boolean type and values <stdbool.h>

```
bool
true
false
__bool_true_false_are_defined
```

B.16 Common definitions <stddef.h>

```
ptrdiff_t     size_t       wchar_t      NULL

offsetof(type, member-designator)
```

B.17 Integer types <stdint.h>

```
int8_t             INT32_MIN          UINT_FAST8_MAX
int16_t            INT64_MIN          UINT_FAST16_MAX
int32_t            INT8_MAX           UINT_FAST32_MAX
int64_t            INT16_MAX          UINT_FAST64_MAX
uint8_t            INT32_MAX          INTPTR_MIN
uint16_t           INT64_MAX          INTPTR_MAX
uint32_t           UINT8_MAX          UINTPTR_MAX
uint64_t           UINT16_MAX         INTMAX_MIN
int_least8_t       UINT32_MAX         INTMAX_MAX
int_least16_t      UINT64_MAX         UINTMAX_MAX
int_least32_t      INT_LEAST8_MIN     PTRDIFF_MIN
int_least64_t      INT_LEAST16_MIN    PTRDIFF_MAX
uint_least8_t      INT_LEAST32_MIN    SIG_ATOMIC_MIN
uint_least16_t     INT_LEAST64_MIN    SIG_ATOMIC_MAX
uint_least32_t     INT_LEAST8_MAX     SIZE_MAX
uint_least64_t     INT_LEAST16_MAX    WCHAR_MIN
int_fast8_t        INT_LEAST32_MAX    WCHAR_MAX
int_fast16_t       INT_LEAST64_MAX    WINT_MIN
int_fast32_t       UINT_LEAST8_MAX    WINT_MAX
int_fast64_t       UINT_LEAST16_MAX   INT8_C(value)
uint_fast8_t       UINT_LEAST32_MAX   INT16_C(value)
uint_fast16_t      UINT_LEAST64_MAX   INT32_C(value)
uint_fast32_t      INT_FAST8_MIN      INT64_C(value)
uint_fast64_t      INT_FAST16_MIN     UINT8_C(value)
intptr_t           INT_FAST32_MIN     UINT16_C(value)
uintptr_t          INT_FAST64_MIN     UINT32_C(value)
intmax_t           INT_FAST8_MAX      UINT64_C(value)
uintmax_t          INT_FAST16_MAX     INTMAX_C(value)
INT8_MIN           INT_FAST32_MAX     UINTMAX_C(value)
INT16_MIN          INT_FAST64_MAX
```

B.18 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);
char *gets(char *s);
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);
```

B.19 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 *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void abort(void);
int atexit(void (*func)(void));
void exit(int status);
char *getenv(const char *name);
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);
```

B.20 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);
```

B.21 Type-generic math <tgmath.h>

```
acos          sqrt          fmod          nearbyint
asin          fabs          frexp         nextafter
atan          atan2         tgamma        nextafterx
acosh         cbrt          hypot         remainder
asinh         ceil          ilogb         remquo
atanh         copysign      ldexp         rint
cos           erf           lgamma        round
sin           erfc          llrint        scalbn
tan           exp2          llround       scalbln
cosh          expm1         log10         trunc
sinh          fdim          log1p         carg
tanh          floor         log2          cimag
exp           fma           logb          conj
log           fmax          lrint         cproj
pow           fmin          lround        creal
```

B.22 Date and time <time.h>

```
NULL                _LOCALTIME        time_t
CLOCKS_PER_SEC      size_t            struct tm
_NO_LEAP_SECONDS    clock_t           struct tmx
```

```
clock_t clock(void);
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm *timeptr);
time_t mkxtime(struct tmx *timeptr);
time_t time(time_t *timer);
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);
size_t strfxtime(char * restrict s,
```

```
        size_t maxsize,
        const char * restrict format,
        const struct tmx * restrict timeptr);
struct tmx *zonetime(const time_t *timer);
```

B.23 Extended multibyte and wide-character utilities `<wchar.h>`

```
wchar_t       wint_t        NULL          WEOF
size_t        struct tm     WCHAR_MAX
mbstate_t     struct tmx    WCHAR_MIN
```

```
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(FILE * restrict stream,
        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 *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);
wchar_t *wcschr(const wchar_t *s, wchar_t c);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
size_t wcslen(const wchar_t *s);
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);
int wmemcmp(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 *wmemset(wchar_t *s, wchar_t c, size_t n);
size_t wcsftime(wchar_t *s, size_t maxsize,
     const wchar_t *format, const struct tm *timeptr);
size_t wcsfxtime(wchar_t *s, size_t maxsize,
     const wchar_t *format, const struct tmx *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);
```

B.24 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 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:

— The call to a function, after the arguments have been evaluated (6.5.2.2).

— The end of the first operand of the following operators: logical AND **&&** (6.5.13); logical OR **||** (6.5.14); conditional **?** (6.5.15); comma **,** (6.5.17).

— The end of a full declarator: declarators (6.7.5);

— The end of a full expression: an initializer (6.7.8); 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 three expressions of a **for** statement (6.8.5.3); the 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 (7.19.6, 7.24.2).

— 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 (7.20.5).

Annex D
(informative)

Formal model of sequence points

D.1 Introduction

1 This annex defines a formal algorithm for determining whether any given expression or set of expressions meets the requirements of 6.5 and therefore whether or not it involves unspecified or undefined behavior.

2 An implementation is not required to follow the model defined in this annex. In particular, the model assumes serial evaluation, but parallel evaluation is equally valid. However, an implementation should ensure — for all expressions that are not determined to be undefined in this model — that expressions are evaluated in a way that will yield one of the results specified by this model, and that each volatile object is accessed exactly the number of times determined by the model.

D.2 Basic concepts

1 This model operates in terms of ‘‘events’’. An expression or set of expressions give rise to a number of events and some constraints on their order of occurrence. All possible arrangements that meet those constraints are then examined. If any of these arrangements involves undefined or unspecified behavior then so does the expression or set of expressions.

D.2.1 Events

1 An event is one of:

— reading the value of a byte,

— writing a value to a byte,

— designation of a byte in an object,

— a function call,

— a sequence point.

2 A ‘‘write’’ event does not mean that the value is stored immediately, but that it will be stored before the next sequence point. The model automatically takes this into account.

D.2.2 Function calls

1 Where an expression involves a function call, that call is "atomic" for the purposes of this model. There must be a sequence point immediately before (see 6.5.2.2) and after each function call (either because it ends in a full expression, or because it is required by 7.1.4), and so it can be seen that the effects of the call — for the purposes of the surrounding expression — can be determined purely by the read and write events involved in it, ignoring their ordering. These events cannot be interspersed with events from outside the call. Therefore this model treats a function call as being a sequence point.

D.2.3 Short circuits and optimization

1 The `&&`, `||`, and `?:` operators are "short circuiting" — the way in which an expression containing them is evaluated depends on the operands. The model described in this annex handles this.

2 Other expressions can sometimes be optimized, for example because the result of an operator can be determined from only some of the operands; the most obvious case is multiplication by constant zero. Not only does this model require the other operand be evaluated in this case, but all the implications of this model must continue to hold even where an optimizing implementation could store values in objects earlier than the model would allow.[285)]

D.2.4 Notation

1 Events are written as follows:

R(*a*)	read the byte at address *a*;
W(*a*)	write the byte at address *a*;
L(*a*)	an lvalue designating the byte at address *a*;
F(*e*)	the call to the function designated by *e*;
S	a sequence point;
D	a dummy event with no effect.

2 For L, R, and W events, the address *a* may also represent a bit-field within a byte in certain circumstances; this is written *a:bf* when it occurs.

3 If it is necessary to distinguish separate events of the same kind, this is done by placing a number or name in braces after the event (such as "R(*a*, *n*){5}" or "S{pre}").

4 In addition, the following notation is used:

$X < Y$	for each event P in those indicated by X, and each event Q in those indicated by Y, P must occur before Q;

285) Of course, provided the relevant side-effects take place as if the model were followed, any optimization that yields a valid result (as specified in D.1) is permitted.

R(*a*, *n*)	R(*a*), R(*a* + 1), ... , R(*a* + *n* − 1)
W(*a*, *n*)	W(*a*), W(*a* + 1), ... , W(*a* + *n* − 1)
L(*a*, *n*)	L(*a*), L(*a* + 1), ... , L(*a* + *n* − 1)
E(*e*)	the events and constraints of expression *e*;
V(*e*)	the events and constraints of expression *e*, but with each L(*a*) event replaced by a (different) dummy event D;
Z(*e*)	the size (in bytes) of an object with the same type as the expression *e* (or 1 if *e* has function type).

D.3 Operation of the model

1 The model is operated in the following stages:

— transforming each expression,

— identifying events and constraints,

— analysis of all possible orders of events.

2 Where more than one expression is involved, each is processed through the first two stages separately, and then combined as given in D.4 before the last stage.

D.3.1 Transformation

1 The first stage of analysis of an expression is to transform it to a canonical form. To do this:

each expression of the form		is replaced by the form
e`->`*field*		`(*(`*e*`)).`*field*
e1`[`*e2*`]`		`*((`*e1*`)+(`*e2*`))`
`&*`*e*		*e*
e1 `&&` *e2*	where *e1* is zero	`(`*e1*`)`
	where *e1* is nonzero	`((`*e1*`) , (`*e2*`))`
e1 `||` *e2*	where *e1* is zero	`((`*e1*`) , (`*e2*`))`
	where *e1* is nonzero	`(`*e1*`)`
e1 `?` *e2* `:` *e3*	where *e1* is zero	`((`*e1*`) , (`*e3*`))`
	where *e1* is nonzero	`((`*e1*`) , (`*e2*`))`

2 Note that the `&&`, `||`, and `?:` operators require multiple analyses to be made; the expression involves undefined behavior in those situations where the corresponding case does; if the value of e1 is unspecified, the expression involves undefined behavior if either alternative does.

3 Finally, wherever 6.5 requires an lvalue not of array type to be converted to the value stored in the designated object, replace the lvalue expression *e* by the expression `$`*e* and where it requires an lvalue of array type to be converted to a pointer to the first element of

the array, or an lvalue of function type to be converted to a pointer to the function, replace it by the expression `@`*e* (where `$` and `@` are notional operators introduced by the model).

4 **D.3.2 Event identification**

1 Each sub-expression gives rise to a set of events and constraints on events as follows.

2 A constant or string literal does not give rise to any events. A parenthesized expression gives rise to the same events and constraints that the unparenthesized version would.

3 An identifier `x` gives rise to L(*a*, Z(`x`)), where *a* is the address of `x`. This includes the case where `x` has an array type. However, an identifier with function type does not give rise to any events. If `x` has register storage class, it still has a notional address even though a program cannot determine that address using the `&` or `@` operators.

4 A compound literal `(`*type*`){`*e1*`, ` *e2*`, ` ...`}` gives rise to L(*a*, Z(*type*)) plus the constraints E(*e1*) < L(*a*, Z(*type*)), E(*e2*) < L(*a*, Z(*type*)), etc., where *a* is the address of the object generated by the compound literal.

5 A type name with variably modified type and containing the expressions *e1*, *e2*, etc. (in array declarators) gives rise to all the events and constraints from V(*e1*), V(*e2*), etc. (with no constraints between these).

6 All other expressions give rise to events and constraints as follows:

Expression	Events and constraints
$*e*	E(*e*), but replacing each L(*a*) by R(*a*).
@*e*	V(*e*).
e0(*e1*, *e2*, ...)	E(*e0*), E(*e1*), E(*e2*), etc., plus new F(*e0*), E(*e0*) < F(*e0*), E(*e1*) < F(*e0*), E(*e2*) < F(*e0*), etc.
e.*field*	E(*e*), but if this contains L(*a*, Z(*e*)) for some address *a* then that is replaced by L(*a* + *b*, Z(*e.field*)), where *b* is the offset of the field in the structure or union type. If **e** has structure (not union) type and the field is a bit-field, the event identifies only that bit-field within the byte(s).
++*e* **--***e* *e***++** *e***--**	E(*e*), but replacing each L(*a*) by the pair [R(*a*), W(*a*)], with the constraint R(*a*) < W(*a*) for each such pair.
&*e*	V(*e*).
*****e*	E(*e*), plus new L(*a*, Z(*****e*)), where *a* is the address pointed to, and the constraint E(*e*) < L(*a*, Z(*****e*)).
sizeof *e* **sizeof (***type***)**	V(*e*) or V(*type*) if *e* or *type* has variably modified type, otherwise none.
+*e* **-***e* **!***e* **~***e*	E(*e*).
(*type***)***e*	V(*e*), E(*type*).
e1 **<***op***>** *e2*	E(*e1*), E(*e2*).
*e1***,** *e2*	E(*e1*), E(*e2*), plus new S, E(*e1*) < S, S < E(*e2*).
e1 **=** *e2*	E(*e1*), but replacing each L(*a*) by W(*a*), E(*e2*), E(*e2*) < W(*a*) for each W(*a*) created by the replacement.
e1 **<***op***=>** *e2*	E(*e1*), but replacing each L(*a*) by the pair [R(*a*), W(*a*)], E(*e2*), and for each W(*a*) created by the replacement the constraints [E(*e2*), R(*a*)] < W(*a*) where the R(*a*) is the corresponding event created by the replacement.

<*op***>** is any of *** / % + - << >> < > <= >= == != & ^ |**

<*op***=>** is any of ***= /= %= += -= <<= >>= &= ^= |=**

D.3.3 Event analysis

1 When all events and constraints in the expression have been determined, every possible arrangement of the remaining events[286] that obeys the constraints is considered.

2 If an arrangement contains W(a) followed by either R(a) or W(a) (for the same address a) without an intervening S or F event, the expression involves undefined behavior.

3 Otherwise, if two different arrangements would have different effects then it is unspecified which results.[287]

4 If an R(a) event applies to a volatile object, this is an access to the object. If there is more than one R(a) event for the same volatile object, the model shows which possible orderings can apply to the accesses for each purpose; if the value of the expression depends on the order of these accesses and the model permits more than one order, which order is used is unspecified.

D.4 Application

D.4.1 Expressions

1 For each expression statement, and for each full expression in a selection, iteration, or jump statement, the expression is considered separately in its normal place in the sequence of execution.

2 For each array declarator involving a variably modified type, and for each initializer list, the events of all the individual expressions are considered together with no constraints applying between events from two different full expressions.

D.4.2 Signals

1 If program execution is interrupted by a signal where the handler returns to the caller, the behavior is as if an F event for the signal handler were inserted at an unspecified place in the sequence of events forming the full expression or array declarator being evaluated at the time of the signal.

---

286) There will be no L(a) events remaining, since they represent lvalues and these will have been made the operand of some operator, possibly a **$** operator.

287) This will occur if one arrangement contains W(a){1} followed by either R(a){2} or W(a){3} and the other arrangement contains either R(a){2} or W(a){3} followed by W(a){1} (in each case possibly with other events intervening).

D.4.3 Floating-point environment

1 For the purposes of this model, the floating-point exception flags constitute a single object and the remainder of the floating-point environment another single object. Any W event to the latter implies a W event to the former with the same constraints and with no constraint between the two.

2 If the state of the **FENV_ACCESS** pragma is on, then any operation that sets a floating-point exception flag implies a W event for the floating-point exception flags ("WX") with the following constraints:

Operations	Constraints
$*e* **+***e* **−***e* **!***e* **~***e* *e1* **<***op***>** *e2*	WX occurs after every event associated with the sub-expression
++*e* **−−***e* *e***++** *e***−−** *e1* **=** *e2* *e1* **<***op=***>** *e2*	WX has the same constraints as the W events generated in the analysis of the sub-expression, but may occur before any or all of those W events.

3 Notwithstanding D.3.3, if an arrangement contains more than one W event for the floating-point exception flags without an intervening S or F event, the behavior is not undefined but it is unspecified which of the flags associated with the W events will be set (though at least one of those flags will be set).[288)]

D.5 Examples

1 In these examples the type **int** and all pointer types have a size of 1, **double** has a size of 3, and addresses are in the range 1000 to 1999. In each case it is the final line that is being analyzed.

2 EXAMPLE 1

```
int x, y, z;
x = y + z;
```

3 The canonical form is:

```
x = $y + $z
```

4 The events and constraints identified by each sub-expression are:

288) If it were undefined to write twice to the flags between sequence points, it would not be practical to use the facilities of the **<fenv.h>** header in complicated expressions.

`x`: L(1000){1}
`y`: L(1234){2}
`z`: L(1666){3}
`$y`: R(1234){4}
`$z`: R(1666){5}
`+`: {4}, {5}
`=`: W(1000){6}, {4}, {5}, [{4}, {5}] < {6}

5 Since each address only appears in one R or W event, there is no undefined behavior.

6 EXAMPLE 2

```
int z, y;
x = y++;      // already in canonical form
```

7 The events and constraints identified by each sub-expression are:

`x`: L(1000){1}
`y`: L(1234){2}
`y++`: R(1234){3}, W(1234){4}, {3} < {4}
`=`: W(1000){5}, {3}, {4}, {3} < {4}, [{3}, {4}] < {5}

8 There is only one possible ordering:

R(1234) : W(1234) : W(1000)

9 Again, each address appears in only one event, so clearly there is no undefined behavior.

10 EXAMPLE 3

```
int x;
x = ++x;      // already in canonical form
```

11 The events and constraints identified by each sub-expression are:

`x`: L(1000){1} [the `x` that is the left operand of `=`]
`x`: L(1000){2} [the `x` that is the operand of `++`]
`++x`: R(1000){3}, W(1000){4}, {3} < {4}
`=`: W(1000){5}, {3}, {4}, {3} < {4}, [{3}, {4}] < {5}

12 Even though there is still only one possible ordering, this time the same address appears in two separate W events without an intervening sequence point, and so the behavior is undefined.

13 EXAMPLE 4

```
int x;
x += x * x;
```

14 The canonical form is:

```
x += $x * $x;
```

15 The events and constraints identified by each sub-expression are:

x:	L(1000){1}	[the **x** that is the left operand of **=**]
x:	L(1000){2}	[the **x** that is the left operand of *****]
x:	L(1000){3}	[the **x** that is the right operand of *****]
$x:	R(1000){4}	[derived from {2}]
$x:	R(1000){5}	[derived from {3}]
*****:	{4}, {5}	
+=:	R(1000){6}, W(1000){7}, {4}, {5}, [{4}, {5}, {6}] < {7}	

16 The address 1000 appears in all four of the remaining events, and this time there are six permitted orderings but none of the 6 involve undefined behavior, and so the expression is valid. If x had volatile-qualified type, it would be unspecified which of the three reads would be used for each of the three operands of the (implied) **x + x * x**.

17 EXAMPLE 5

```
int x;
extern int f(int);
x = f(x++);
```

18 The canonical form is:

```
x = (@f)(x++)
```

19 The events and constraints identified by each sub-expression are:

x:	L(1000){1}	[the **x** that is the left operand of **=**]
x:	L(1000){2}	[the **x** that is operand of **++**]
x++:	R(1000){3}, W(1000){4}, {3} < {4}	
f:	none	
@f:	none	
call:	{3}, {4}, F(**f**){5}, {3} < {4}, [{3}, {4}] < {5}	
=:	W(1000){6}, {3}, {4}, {5}, {3} < {4}, [{3}, {4}] < {5}, [{3}, {4}, {5}] < {6}	

20 The four events have one possible ordering:

R(1000) : W(1000) : F(**f**) : W(1000)

21 It is therefore clear that **x** is incremented before the function call starts execution, that the result of the function is stored in **x**, and that the expression is valid. If the function has some other form of access to **x**, it will see the incremented value, and any value it stores in **x** will be lost.

22 If the expression is replaced by:

```
x = 2 * f(x++);
```

the analysis is effectively the same — the multiplication by a constant adds no new events. And if it is replaced by:

```
x = 0 * f(x++);
```

the analysis continues to remain the same; an optimizing implementation can determine that **x** is set to 0, but it must still make the incremented value of **x** available to the function.

23 EXAMPLE 6

```
int x, y;
(x=y) + x;
```

24 The canonical form is:

```
(x=$y)+$x
```

25 The events and constraints identified by each sub-expression are:

x:	L(1000){1}	[the **x** that is the left operand of **=**]
y:	L(1234){2}	
x:	L(1000){3}	[the **x** that is the right operand of **+**]
$y:	R(1234){4}	
=:	W(1000){5}, {4}, {4} < {5}	
$x:	R(1000){6}	
+:	{4}, {5}, {6}, {4} < {5}	

26 There are three possible orderings:

{4} : {5} : {6}	R(1234) : W(1000) : R(1000)
{4} : {6} : {5}	R(1234) : R(1000) : W(1000)
{6} : {4} : {5}	R(1000) : R(1234) : W(1000)

27 The first of the three involves reading address 1000 after writing it — without an intervening sequence point — and so the expression is undefined.

28 EXAMPLE 7

```
int x, y, z;
(x=y) + (x=z);
```

29 The canonical form is:

```
(x=$y)+(x=$z)
```

30 The events and constraints identified by each sub-expression are:

x:	L(1000){1}	[the **x** that is set to **y**]
y:	L(1234){2}	
x:	L(1000){3}	[the **x** that is set to **z**]
z:	L(1666){4}	
$y:	R(1234){5}	
x=$y:	W(1000){6}, {5}, {5} < {6}	
$z:	R(1666){7}	
x=$z:	W(1000){8}, {7}, {7} < {8}	
+:	{5}, {6}, {7}, {8}, {5} < {6}, {7} < {8}	

31 There are six possible orderings, but all involve {6} and {8} not separated by a sequence point, and so the expression is undefined.

32 EXAMPLE 8 To demonstrate the rules for arrays:

```
double x [5];
int y = 3;
x[y] /= (double) &x[y];
```

33 The canonical form is:

```
*(@x+$y) /= (double)(@x+$y);
```

34 The events and constraints identified by each sub-expression are:

`x`:	L(1000,15){1}	[the `x` to the left of the `=`]
`y`:	L(1666){2}	[the `y` to the left of the `=`]
`x`:	L(1000,15){3}	[the `x` to the right of the `=`]
`y`:	L(1666){4}	[the `y` to the right of the `=`]
`@x`:	D{5}	[derived from {1}]
`$y`:	R(1666){6}	[derived from {2}]
`+`:	{5}, {6}	
`x[y]`:	{5}, {6}, L(1009,3){7}, [{5}, {6}] < {7}	
`@x`:	D{8}	[derived from {3}]
`$y`:	R(1666){9}	[derived from {4}]
`+`:	{8}, {9}	
`double`:	none	
cast:	{8}, {9}	
`/=`:	{5}, {6}, R(1009,3){10}, W(1009,3){11}, {5} < {10}, {5} < {11}, {6} < {10}, {6} < {11}, {8}, {9}, [{8}, {9}, {10}] < {11}	

35 The only purpose of D events is to preserve ordering in cases like [{22} < D, D < {23}] and so {5} and {8} can be ignored. This leaves the events:

R(1666){6}, R(1666){9}, R(1009,3){10}, W(1009,3){11}

and the constraints:

{6} < {10}, {6} < {11}, {9} < {11}, {10} < {11}

36 This permits three orderings:

R(1666){6} : R(1009,3) : R(1666){9} : W(1009,3)
R(1666){6} : R(1666){9} : R(1009,3) : W(1009,3)
R(1666){9} : R(1666){6} : R(1009,3) : W(1009,3)

37 None of these involve undefined behavior.

38 EXAMPLE 9 To demonstrate the rules for structures:

```
int x;
struct s { double p; int q; double r; } y;
x = y.q;
```

39 The canonical form is:

```
x = $(y.q)
```

40 The events and constraints identified by each sub-expression are:

`x`:	L(1000){1}
`y`:	L(1230,7){2}
`y.q`:	L(1233,1){3}
`$(y.q)`:	R(1233){4}
`=`:	W(1000){5}, {4}, {4} < {5}

41 EXAMPLE 10

```
struct s { double p; int q; int r; } *x, y;
x = &y;
x->q = x->r;
```

42 The canonical form is:

```
(*$x).q = $((*$x).r)
```

43 The events and constraints identified by each sub-expression are:

x:	L(1000){1}	[part of the left operand of the **=**]
x:	L(1000){2}	[part of the right operand of the **=**]
$x:	R(1000){3}	[derived from {1}]
***$x**:	{3}, L(1220,5){4}, {3} < {4}	
x->q:	{3}, L(1223,1){5}, {3} < {5}	
$x:	R(1000){6}	[derived from {2}]
***$x**:	{6}, L(1220,5){7}, {6} < {7}	
x->r:	{6}, L(1224,1){8}, {6} < {8}	
$(x->r):	{6}, R(1224,1){9}, {6} < {9}	
=:	{3}, W(1223,1){10}, {3} < {10}, {6}, {9}, {6} < {9}, [{6}, {9}] < {10}	

44 These constraints allow three orderings:

{3} : {6} : {9} : {10}
{6} : {3} : {9} : {10}
{6} : {9} : {3} : {10}

and it is clear that all of these are valid.

45 EXAMPLE 11 To illustrate the rules for bit-fields:

```
struct { int x : 10; int y : 3; } s;
s.y = 4;
s.x = s.y++;
```

46 This is already in canonical form.

47 The events and constraints identified by each sub-expression are:

s:	L(1000){1}	[part of **s.x**]
s:	L(1000){2}	[part of **s.y**]
s.x:	L(1000:0–9){3}	[refers only to the bits of field **x**]
s.y:	L(1000:10–12){4}	[refers only to the bits of field **y**]
s.y++:	R(1000:10–12){5}, W(1000:10–12){6}, {5} < {6}	
=:	W(1000:0–9){6}, {4}, {5}, {4} < {5}, [{4}, {5}] < {6}	

48 There is only one possible ordering:

R(1000:10–12), W(1000:10–12), W(1000:0–9)

49 Since the two W events identify different bit-fields, there is no undefined behavior.

50 However, if the type was a union instead of a structure, there would be no bit-fields in the events, and the final ordering would be:

R(1000), W(1000), W(1000)

which of course involves undefined behavior.

51 EXAMPLE 12

```
int x;
x++ && x--;
```

52 This expression has two canonical forms, depending on whether **x** is or is not initially zero:

```
x++            // x is initially 0
x++ , x--      // x is not initially 0
```

53 In this case it is clear that only the latter needs to be examined further, but this might not be the case in more complex expression. Doing so, the events and constraints identified by each sub-expression are:

x:	L(1000){1}	[the **x** in **x++**]
x:	L(1000){2}	[the **x** in **x--**]
x++:	R(1000){3}, W(1000){4}, {3} < {4}	
x--:	R(1000){5}, W(1000){6}, {5} < {6}	
,:	{3}, {4}, {3} < {4}, {5}, {6}, {5} < {6}, S{7}, [{3}, {4}] < {7}, {7} < [{5}, {6}]	

54 There is only one possible ordering, and this has two W events to the same address, but these are separated by an S event and so the expression is valid. It is clear that the decrement must apply to the incremented variable.

55 EXAMPLE 13

```
int x, y;
x++ * y++ ? x-- : y--;
```

56 Again there are two canonical forms:

```
x++ * y++ , x--    // x and y both initially nonzero
x++ * y++ , y--    // either x or y initially zero
```

57 In each case the analysis follows the lines above. The first case yields:

R(1000){1}, W(1000){2}, R(1234){3}, W(1234){4}, S{5}, R(1000){6}, W(1000){7},
{1} < {2}, {1} < {5}, {2} < {5}, {3} < {4}, {3} < {5}, {4} < {5}, {5} < {6}, {5} < {7}, {6} < {7}

58 There are six possible orderings, but all place a sequence point between the first and second writes to **x**.

59 EXAMPLE 14

```
int x[2], *y;
y = x;
*y = f(y++);
```

60 The canonical form is:

```
*$y = (@f)(y++)
```

61 The events and constraints identified by each sub-expression are:

`y`:	L(1234){1}	[the **`y`** on the left of the **`=`**]
`y`:	L(1234){2}	[the **`y`** that is the operand of **`y++`**]
`y++`:	R(1234){3}, W(1234){4}, {3} < {4}	
`f`:	none	
`@f`:	none	
call:	{3}, {4}, {3} < {4}, F(**`f`**){5}, [{3}, {4}] < {5}	
`$y`:	R(1234){6}	[derived from {1}]
`*$y`:	{6}, L(*X*){7}, {6} < {7}	
`=`:	{6}, W(*X*){8}, {6} < {8}, {3}, {4}, {5}, {3} < {4}, [{3}, {4}] < {5}, [{3}, {4}, {5}] < {8}	

where *X* is the location pointed to by **`y`** at the point of the **`*`** operator.

62 One possible ordering is:

R(1234){3} : W(1234){4} : R(1234){6} : F(**`f`**){5} : W(*X*){8}

and therefore the expression involves undefined behavior.

63 EXAMPLE 15

```
int x[2], y;
y = 0;
x[y] = f(y++);
```

64 The canonical form is:

```
*(@x+$y) = (@f)(y++)
```

65 The events and constraints identified by each sub-expression are:

`x`:	L(1000,2){1}	
`y`:	L(1234){2}	[the **`y`** on the left of the **`=`**]
`y`:	L(1234){3}	[the **`y`** that is the operand of **`y++`**]
`@x`:	D{4}	
`$y`:	R(1234){5}	[derived from {1}]
`+`:	{4}, {5}	
`x[y]`:	{4}, {5}, L(*X*){6}, [{4}, {5}] < {6} where *X* is 1000 plus the value determined at {5}	
`y++`:	R(1234){7}, W(1234){8}, {7} < {8}	
`f`:	none	
`@f`:	none	
call:	{7}, {8}, {7} < {8}, F(**`f`**){9}, [{7}, {8}] < {9}	
`=`:	{4}, {5}, W(*X*){10}, [{4}, {5}] < {10}, {7}, {8}, {9}, {7} < {8}, {8} < {9}, {7} < {9}, [{7}, {8}, {9}] < {10}	

66 One possible ordering is:

D{4} : R(1234){7} : W(1234){8} : R(1234){5} : F(**`f`**){9} : W(*X*){10}

and therefore the expression involves undefined behavior.

67 EXAMPLE 16

```
int x = 5;
int a [x][x++];
```

68 There are two expressions to be considered, and their canonical forms are:

```
$x
x++
```

69 The first generates the event:

R(1000){1}

and the second the events and constraints:

R(1000){2}, W(1000){3}, {2} < {3}

70 There are no constraints between {1} and the other two events, and so they can occur in any of three orders. One of these arrangements puts W(1000) before R(1000) and so the declaration as a whole involves undefined behavior.

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 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_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 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__`** conforms to the specifications in this annex. Where a binding between the C language and IEC 60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless stated otherwise.

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,[289] 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.[290]

Recommended practice

2 The **`long double`** type should match an IEC 60559 extended format.

289) *Extended* is IEC 60559's double-extended data format. Extended refers to both the common 80-bit and quadruple 128-bit IEC 60559 formats.

290) A non-IEC 60559 **`long double`** type is required to provide infinity and NaNs, as its values include all **`double`** values.

F.2.1 Infinities, signed zeros, and NaNs

1 This specification does not define the behavior of signaling NaNs.[291] 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** function in **<math.h>** provides the IEC 60559 square root operation.

— The **remainder** function in **<math.h>** provides the IEC 60559 remainder operation. The **remquo** function in **<math.h>** provides the same operation but with additional information.

— The **rint** function in **<math.h>** provides the IEC 60559 operation that rounds a floating-point number to an integer value (in the same precision). The C **nearbyint** function in **<math.h>** provides the nearbyinteger function recommended in the Appendix to IEEE standard 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**, **fprintf**, **fscanf**, and related library functions in **<stdlib.h>**, **<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 IEEE standard 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**,

---

291) Since NaNs created by IEC 60559 operations are always quiet, quiet NaNs (along with infinities) are sufficient for closure of the arithmetic.

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 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 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 (**FE_TONEAREST**, **FE_UPWARD**, **FE_DOWNWARD**, **FE_TOWARDZERO**) also in **<fenv.h>**.

— 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** function in **<math.h>** provides the copysign function recommended in the Appendix to IEC 60559.

— The unary minus (**-**) operator provides the minus (−) operation recommended in the Appendix to IEC 60559.

— The **scalbn** and **scalbln** functions in **<math.h>** provides the scalb function recommended in the Appendix to IEC 60559.

— The **logb** function in **<math.h>** provides the logb function recommended in the Appendix to IEC 60559, but following the newer specifications in IEEE 854.

— The **nextafter** and **nextafterx** 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 **fpclassify** does not distinguish signaling from quiet NaNs).

F.4 Floating to integer conversion

1 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* exception is raised and the resulting value is unspecified. Whether conversion of non-integer floating values whose integral part is within the range of the integer type raises the inexact exception is unspecified.[292)]

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.[293)]

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.

F.6 Contracted expressions

1 A contracted expression 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 exceptions in a manner generally consistent with the basic arithmetic operations. A contracted expression should deliver the same value as its uncontracted counterpart, else should be correctly rounded (once).

F.7 Environment

1 The floating-point environment defined in **<fenv.h>** includes the IEC 60559 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.[294)]

---

292) ANSI/IEEE 854, but not IEC 60559 (ANSI/IEEE 754), directly specifies that floating-to-integer conversions raise the *inexact* 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 exception. See **rint**, **lrint**, **llrint**, and **nearbyint** in **<math.h>**.

293) 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 19 and 15, respectively, for these formats.)

294) This specification does not require dynamic rounding precision nor trap enablement modes.

F.7.1 Environment management

1 IEC 60559 requires that floating-point operations implicitly raise 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.[295)]

F.7.2 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 exceptions.

Recommended practice

2 The implementation should produce a diagnostic message for each translation-time floating-point exception, other than inexact;[296)] the implementation should then proceed with the translation of the program.

F.7.3 Execution

1 At program startup the floating-point environment is initialized as prescribed by IEC 60559:

— All 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 exceptions.

---

295) If the state for the **`FENV_ACCESS`** pragma is *off*, the implementation is free to assume the modes will be the default ones and the flags will not be tested, which allows certain optimizations (see F.8).

296) 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 exceptions (even where the state of the **`FENV_ACCESS`** pragma is *on*). Library functions, for example **`strtod`**, provide execution-time conversion of numeric strings.

F.7.4 Constant expressions

1 An arithmetic constant expression of floating type, other than one in an initializer for an object that has static storage duration, is evaluated (as if) during execution; thus, it is affected by any operative modes and raises exceptions as required by IEC 60559 (provided the state for the **FENV_ACCESS** pragma is *on*).[297)]

2 EXAMPLE

```
#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) exceptions. On the other hand, for the three automatic initializations the invalid division occurs at execution time.

F.7.5 Initialization

1 All computation for automatic initialization is done (as if) at execution time; thus, it is affected by any operative modes and raises 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 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
      /* ... */
}
```

297) 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 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;
```

3 The static initialization of **v** raises no (execution-time) 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 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 exception is raised.[298)] The automatic initialization of **z** entails execution-time conversion, but not to a narrower format, so no exception is raised. Note that the conversions of the floating constants **1.1e75** and **1.1e75f** to their internal representations occur at translation time in all cases.

F.7.6 Changing the environment

1 Operations defined in 6.5 and functions and macros defined for the standard libraries change flags and 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 exceptions for atomic operations (namely *overflow* and *inexact*, or *underflow* and *inexact*) then *overflow* or *underflow* is raised before *inexact*.

F.8 Optimization

1 This section identifies code transformations that might subvert IEC 60559-specified behavior, and others that do not.

F.8.1 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

---

298) 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.

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(double x)
{
     /* ... */
     for (i = 0; i < n; i++) x + 1;
     /* ... */
}
```

`x + 1` might raise 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 exceptions. Therefore, between function calls exceptions need not be precise: the actual order and number of occurrences of exceptions (> 1) may vary from what the source code expresses. Thus the preceding loop could be treated as

```
if (0 < n) x + 1;
```

F.8.2 Expression transformations

1	`x / 2` $\leftrightarrow$ `x * 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 * x` and `x / 1` $\rightarrow$ `x`	The expressions `1 * x`, `x / 1`, and `x` are equivalent (on IEC 60559 machines, among others).[299)]
	`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).[300)]

---

299) Strict support for signaling NaNs — not required by this specification — would invalidate these and other transformations that remove arithmetic operators.

300) 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.

x - x → **0.0**	The expressions **x - x** and **0.0** are not equivalent if **x** is a NaN or infinite.
0 * x → **0.0**	The expressions **0 * x** and **0.0** are not equivalent if **x** is a NaN, infinite, or −0.
x + 0 → **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 → **x**	(+0) − (+0) yields −0 when rounding is downward (toward −∞), 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 ↔ **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).

F.8.3 Relational operators

1 **x != x** → false	The statement **x != x** is true if **x** is a NaN.
x == x → true	The statement **x == x** is false if **x** is a NaN.
x < y → **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* 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.8.4 Constant arithmetic

1 The implementation shall honor exceptions raised by execution-time constant arithmetic wherever the state of the **FENV_ACCESS** pragma is *on*. (See F.7.4 and F.7.5.) An operation on constants that raises no 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,[301] and implementations that support dynamic rounding precision modes shall assure further that the result of the operation raises no exception when converted to the semantic type of the operation.

301) $0 - 0$ yields -0 instead of $+0$ just when the rounding direction is downward.

F.9 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 exception flags in a manner consistent with the basic arithmetic operations covered by IEC 60559.

4 The *invalid* and *divide-by-zero* exceptions are raised as specified in subsequent subclauses of this annex.

5 The *overflow* 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.

6 The *underflow* exception is raised whenever a result is tiny (essentially subnormal or zero) and suffers loss of accuracy.[302)]

7 Whether or when the trigonometric, hyperbolic, base-e exponential, base-e logarithmic, error, and log gamma functions raise the *inexact* exception is implementation-defined. For other functions, the *inexact* exception is raised whenever the rounded result is not identical to the mathematical result.

8 Whether the *inexact* exception may be raised when the rounded result actually does equal the mathematical result is implementation-defined. Whether the *underflow* (and *inexact*) exception may be raised when a result is tiny but not inexact is implementation-defined.[303)] Otherwise, as implied by F.7.6, the **<math.h>** functions do not raise spurious exceptions (detectable by the user).

9 Whether the functions honor the rounding direction mode is implementation-defined.

10 Functions with a NaN argument return a NaN result and raise no exception, except where stated otherwise.

11 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.

---

302) IEC 60559 allows different definitions of underflow. They all result in the same values, but differ on when the exception is raised.

303) It is intended that undeserved *underflow* and *inexact* exceptions are raised only if determining inexactness would be too costly.

Recommended practice

12 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 (converted to its parameter type), except perhaps for the sign.

F.9.1 Trigonometric functions

F.9.1.1 The `acos` functions

1 — **`acos(1)`** returns +0.

— **`acos(x)`** returns a NaN and raises the *invalid* exception for |**`x`**| > 1.

F.9.1.2 The `asin` functions

1 — **`asin(±0)`** returns ±0.

— **`asin(x)`** returns a NaN and raises the *invalid* exception for |**`x`**| > 1.

F.9.1.3 The `atan` functions

1 — **`atan(±0)`** returns ±0.

— **`atan(±∞)`** returns $\pm\pi/2$.

F.9.1.4 The `atan2` functions

1 — **`atan2(±0, x)`** returns ±0, for **`x`** > 0.

— **`atan2(±0, +0)`** returns ±0.[304)]

— **`atan2(±0, x)`** returns $\pm\pi$, for **`x`** < 0.

— **`atan2(±0, -0)`** returns $\pm\pi$.

— **`atan2(y, ±0)`** returns $\pi/2$ for **`y`** > 0.

— **`atan2(y, ±0)`** returns $-\pi/2$ for **`y`** < 0.

— **`atan2(±y, ∞)`** returns ±0, for finite **`y`** > 0.

— **`atan2(±∞, x)`** returns $\pm\pi/2$, for finite **`x`**.

— **`atan2(±y, -∞)`** returns $\pm\pi$, for finite **`y`** > 0.

— **`atan2(±∞, ∞)`** returns $\pm\pi/4$.

— **`atan2(±∞, -∞)`** returns $\pm 3\pi/4$.

304) **`atan2(0, 0)`** does not raise the *invalid* exception, nor does **`atan2(y, 0)`** raise the *divide-by-zero* exception.

F.9.1.5 The `cos` functions

1 — `cos(±0)` returns 1.

— `cos(±∞)` returns a NaN and raises the *invalid* exception.

F.9.1.6 The `sin` functions

1 — `sin(±0)` returns ±0.

— `sin(±∞)` returns a NaN and raises the *invalid* exception.

F.9.1.7 The `tan` functions

1 — `tan(±0)` returns ±0.

— `tan(±∞)` returns a NaN and raises the *invalid* exception.

F.9.2 Hyperbolic functions

F.9.2.1 The `acosh` functions

1 — `acosh(1)` returns +0.

— `acosh(+∞)` returns +∞.

— `acosh(x)` returns a NaN and raises the *invalid* exception if **`x`** < 1.

F.9.2.2 The `asinh` functions

1 — `asinh(±0)` returns ±0.

— `asinh(±∞)` returns ±∞.

F.9.2.3 The `atanh` functions

1 — `atanh(±0)` returns ±0.

— `atanh(±1)` returns ±∞ and raises the *divide-by-zero* exception.

— `atanh(x)` returns a NaN and raises the *invalid* exception if |**`x`**| > 1.

F.9.2.4 The `cosh` functions

1 — **`cosh(`**$\pm$**`0)`** returns 1.

— **`cosh(`**$\pm\infty$**`)`** returns $+\infty$.

F.9.2.5 The `sinh` functions

1 — **`sinh(`**$\pm$**`0)`** returns ± 0.

— **`sinh(`**$\pm\infty$**`)`** returns $\pm\infty$.

F.9.2.6 The `tanh` functions

1 — **`tanh(`**$\pm$**`0)`** returns ± 0.

— **`tanh(`**$\pm\infty$**`)`** returns ± 1.

F.9.3 Exponential and logarithmic functions

F.9.3.1 The `exp` functions

1 — **`exp(`**$\pm$**`0)`** returns 1.

— **`exp(+`**∞**`)`** returns $+\infty$.

— **`exp(−`**∞**`)`** returns $+0$.

F.9.3.2 The `exp2` functions

1 — **`exp2(`**$\pm$**`0)`** returns 1.

— **`exp2(+`**∞**`)`** returns $+\infty$.

— **`exp2(−`**∞**`)`** returns $+0$.

F.9.3.3 The `expm1` functions

1 — **`expm1(`**$\pm$**`0)`** returns ± 0.

— **`expm1(+`**∞**`)`** returns $+\infty$.

— **`expm1(−`**∞**`)`** returns -1.

F.9.3.4 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(x, exp)`** stores an unspecified value in the object pointed to by **`exp`** (and returns a NaN) when **`x`** is a NaN.

— **`frexp`** raises no exception.

2 On a binary system, **`frexp`** is equivalent to the comma expression

```
( (*exp = (value == 0) ? 0 :
     (int)(1 + logb(value))), scalbn(value, -(*exp)) )
```

F.9.3.5 The `ilogb` functions

1 No additional requirements.

F.9.3.6 The `ldexp` functions

1 On a binary system, **`ldexp(x, exp)`** is equivalent to

```
scalbn(x, exp)
```

F.9.3.7 The `log` functions

1 — **`log(±0)`** returns −∞ and raises the *divide-by-zero* exception.

— **`log(1)`** returns +0.

— **`log(x)`** returns a NaN and raises the *invalid* exception if **`x`** < 0.

— **`log(+∞)`** returns +∞.

F.9.3.8 The `log10` functions

1 — **`log10(±0)`** returns −∞ and raises the *divide-by-zero* exception.

— **`log10(1)`** returns +0.

— **`log10(x)`** returns a NaN and raises the *invalid* exception if **`x < 0`**.

— **`log10(+∞)`** returns +∞.

F.9.3.9 The **log1p** functions

1 — **log1p(±0)** returns ±0.

— **log1p(−1)** returns −∞ and raises the *divide-by-zero* exception.

— **log1p(x)** returns a NaN and raises the *invalid* exception if **x** < − 1.

— **log1p(+∞)** returns +∞.

F.9.3.10 The **log2** functions

1 — **log2(±0)** returns −∞ and raises the *divide-by-zero* exception.

— **log2(x)** returns a NaN and raises the *invalid* exception if **x** < 0.

— **log2(+∞)** returns +∞.

F.9.3.11 The **logb** functions

1 — **logb(±∞)** returns +∞.

— **logb(±0)** returns −∞ and raises the *divide-by-zero* exception.

F.9.3.12 The **modf** functions

1 — **modf(value, iptr)** returns a result with the same sign as the argument **value**.

— **modf(±∞, iptr)** returns ±0 and stores ±∞ in the object pointed to by **iptr**.

— **modf** of a NaN argument stores a NaN in the object pointed to by **iptr** (and returns a NaN).

2 **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.9.3.13 The `scalbn` and `scalbln` functions

1 — `scalbn(x, n)` returns `x` if `x` is infinite or zero.

— `scalbn(x, 0)` returns `x`.

F.9.4 Power and absolute value functions

F.9.4.1 The `cbrt` functions

1 — `cbrt(±∞)` returns ±∞.

— `cbrt(±0)` returns ±0.

F.9.4.2 The `fabs` functions

1 — `fabs(±0)` returns +0.

— `fabs(±∞)` returns +∞.

F.9.4.3 The `hypot` functions

1 — `hypot(x, y)`, `hypot(y, x)`, and `hypot(x, −y)` are equivalent.

— `hypot(x, y)` returns +∞ if `x` is infinite, even if `y` is a NaN.

— `hypot(x, ±0)` is equivalent to `fabs(x)`.

F.9.4.4 The `pow` functions

1 — `pow(x, ±0)` returns 1 for any `x`, even a NaN.

— `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(+∞, y)` returns +∞ for `y` > 0.

— `pow(+∞, y)` returns +0 for `y` < 0.

— `pow(−∞, y)` returns −∞ for `y` an odd integer > 0.

— `pow(−∞, y)` returns +∞ for `y` > 0 and not an odd integer.

— `pow(−∞, y)` returns −0 for `y` an odd integer < 0.

— `pow(−∞, y)` returns +0 for `y` < 0 and not an odd integer.

— `pow(±1, ±∞)` returns a NaN and raises the *invalid* exception.

— `pow(x, y)` returns a NaN and raises the *invalid* exception for finite `x` < 0 and finite non-integer `y`.

— `pow(±0, y)` returns ±∞ and raises the *divide-by-zero* exception for `y` an odd integer < 0.

— **`pow(±0, y)`** returns +∞ and raises the *divide-by-zero* exception for **`y`** < 0 and not an odd integer.

— **`pow(±0, y)`** returns ±0 for **`y`** an odd integer > 0.

— **`pow(±0, y)`** returns +0 for **`y`** > 0 and not an odd integer.

F.9.4.5 The `sqrt` functions

1 **`sqrt`** is fully specified as a basic arithmetic operation in IEC 60559.

F.9.5 Error and gamma functions

F.9.5.1 The `erf` functions

1 — **`erf(±0)`** returns ±0.

— **`erf(±∞)`** returns ±1.

F.9.5.2 The `erfc` functions

1 — **`erfc(+∞)`** returns +0.

— **`erfc(−∞)`** returns 2.

F.9.5.3 The `lgamma` functions

1 — **`lgamma(+∞)`** returns **+**∞.

— **`lgamma(x)`** returns +∞ and raises the *divide-by-zero* exception if **`x`** is a negative integer or zero.

— **`lgamma(−∞)`** returns **+**∞.

F.9.5.4 The `tgamma` functions

1 — **`tgamma(+∞)`** returns +∞.

— **`tgamma(x)`** returns a NaN and raises the *invalid* exception if **`x`** is a negative integer or zero.

— **`tgamma(−∞)`** returns a NaN and raises the *invalid* exception.

F.9.6 Nearest integer functions

F.9.6.1 The `ceil` functions

1 — `ceil(x)` returns `x` if `x` is ±∞ or ±0.

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;
}
```

F.9.6.2 The `floor` functions

1 — `floor(x)` returns `x` if `x` is ±∞ or ±0.

See the sample implementation for `ceil` in F.9.6.1.

F.9.6.3 The `nearbyint` functions

1 The `nearbyint` functions use IEC 60559 rounding according to the current rounding direction. They do not raise the *inexact* exception if the result differs in value from the argument.

F.9.6.4 The `rint` functions

1 The `rint` functions differ from the `nearbyint` functions only in that they do raise the *inexact* exception if the result differs in value from the argument.

— `rint(±0)` returns ±0 (for all rounding directions).

— `rint(±∞)` returns ±∞ (for all rounding directions).

F.9.6.5 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* exception is raised. When they raise no other exception and the result differs from the argument, they raise the *inexact* exception.

F.9.6.6 The **round** functions

1 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* exception for non-integer numeric arguments, as this implementation does.

F.9.6.7 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 may (but need not) raise the *inexact* exception for non-integer arguments that round to within the range of the return type.

F.9.6.8 The **trunc** functions

1 The **trunc** functions use IEC 60559 rounding toward zero (regardless of the current rounding direction).

F.9.7 Remainder functions

F.9.7.1 The `fmod` functions

1 — **`fmod(±0, y)`** returns ±0 if **`y`** is not zero.

— **`fmod(x, y)`** returns a NaN and raises the *invalid* exception if **`x`** is infinite or **`y`** is zero.

— **`fmod(x, ±∞)`** returns **`x`** if **`x`** is not infinite.

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.9.7.2 The `remainder` functions

1 The **`remainder`** functions are fully specified as a basic arithmetic operation in IEC 60559.

F.9.7.3 The `remquo` functions

1 The **`remquo`** functions follow the specifications for the **`remainder`** functions. They have no further specifications special to IEC 60559 implementations.

F.9.8 Manipulation functions

F.9.8.1 The `copysign` functions

1 **`copysign`** is specified in the Appendix to IEC 60559.

F.9.8.2 The **nan** functions

1 All IEC 60559 implementations support quiet NaNs, in all floating formats.

F.9.8.3 The **nextafter** functions

1 — **nextafter(x, y)** raises the *overflow* and *inexact* exceptions if **x** is finite and the function value is infinite.

— **nextafter(x, y)** raises the *underflow* and *inexact* exceptions if the function value is subnormal or zero and **x** ≠ **y**.

F.9.8.4 The **nextafterx** functions

No additional requirements.

F.9.9 Maximum, minimum, and positive difference functions

F.9.9.1 The **fdim** functions

1 No additional requirements.

F.9.9.2 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).

The body of the **fmax** function might be[305)]

```
{ return (isgreaterequal(x, y) ||
      isnan(y)) ? x : y; }
```

F.9.9.3 The **fmin** functions

1 The **fmin** functions are analogous to the **fmax** functions. See F.9.9.2.

F.9.10 Floating point multiply-add

305) 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.9.10.1 The `fma` functions

1 — **`fma(x, y, z)`** computes the sum **`z`** plus the product **`x`** times **`y`**, correctly rounded once.

— **`fma(x, y, z)`** returns a NaN and optionally raises the *invalid* 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* 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* exception if **`x`** times **`y`** is an exact infinity and **`z`** is also an infinity but with the opposite sign.

Annex G
(informative)

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 supports this specification if and only if it defines the macro **`__STD_IEC_559_COMPLEX__`**.

G.2 Types

1 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.

2 For imaginary types, the corresponding real type is given by deleting the keyword **`_Imaginary`** from the type name.

3 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.

4 The *imaginary type domain* comprises the imaginary types.

G.3 Conversions

G.3.1 Imaginary types

1 Conversions among imaginary types follow rules analogous to those for real floating types.

G.3.2 Real and imaginary

1 When a value of imaginary type is converted to a real type, 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.3.3 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.4 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.4.1), but is not completely determined.

G.4.1 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 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 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 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. The `*` and `/` operators satisfy the following infinity properties for all real, imaginary, and complex operands:[306)]

— 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;

— 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 exceptions if appropriate for the calculation of the parts of the result, and may raise spurious 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.5).

306) 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*).

```
#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;
      /* Recover infinities that computed as NaN+iNaN ... */
      if (isnan(x) && isnan(y)) {
            int recalc = 0;
            if  ( isinf(a) || isinf(b) ) { /* z is infinite */
                  /* "Box" the infinity ... */
                  a = copysign(isinf(a) ? 1.0 : 0.0, a);
                  b = copysign(isinf(b) ? 1.0 : 0.0, b);
                  /* Change NaNs in the other factor to 0 ... */
                  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 ... */
                  c = copysign(isinf(c) ? 1.0 : 0.0, c);
                  d = copysign(isinf(d) ? 1.0 : 0.0, d);
                  /* Change NaNs in the other factor to 0 ... */
                  if (isnan(a)) a = copysign(0.0, a);
                  if (isnan(b)) b = copysign(0.0, b);
                  recalc = 1;
            }
            if (!recalc) {
                  /* *Recover infinities from overflow cases ... */
                  if (isinf(ac) || isinf(bd) ||
                        isinf(ad) || isinf(bc)) {
                        /* Change all 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 non-zero/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 (isinf(logbw) &&
                  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.4.2 Additive operators

Semantics

1 If one operand has real type and the other operand has imaginary type, then the result has complex type. If both operands have imaginary type, then the result has imaginary type. (If either operand has complex type, then the result has complex type.)

2 In all cases the result and 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.5 Complex arithmetic **<complex.h>**

1 The macros

```
imaginary
```

and

```
_Imaginary_I
```

are defined, and the macro

```
I
```

is defined to be **`_Imaginary_I`** (7.3).

2 This subclause contains specifications for the **`<complex.h>`** functions that are particularly suited to IEC 60559 implementations.

3 The functions are continuous onto both sides of their branch cuts, taking into account the sign of zero. For example, **`csqrt(-2 ± 0*I) == ±sqrt(2)*I`**.

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 exception flags in a manner consistent with the specifications for real functions in F.9.[307)]

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 exceptions.

307) As noted in G.4.1, a complex value with at least one infinite part is regarded as an infinity even if its other part is a NaN.

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**, **ctan**, and **cpow** 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**)**
cpow(z**,** c**)** = **cexp(**c **clog(**z**))**

8 For the other functions, the following subclauses specify behavior for special cases, including treatment of the *invalid* and *divide-by-zero* 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 specification for the upper half-plane implies the specification for the lower half-plane; if also the function f is either even, $f(-z) = f(z)$, or odd, $f(-z) = -f(z)$, then the specification for the first quadrant implies the specification for the other three quadrants.

9 In the following subclauses, $\mathrm{cis}(y)$ is defined as $\cos(y) + i\sin(y)$.

G.5.1 Trigonometric functions

G.5.1.1 The cacos functions

1
— **cacos(conj(**z**)) = conj(cacos(**z**))**.

— **cacos(**$\pm 0 + i0$**)** returns $\pi/2 - i0$.

— **cacos(**$-\infty + i\infty$**)** returns $3\pi/4 - i\infty$.

— **cacos(**$+\infty + i\infty$**)** returns $\pi/4 - i\infty$.

— **cacos(**$x + i\infty$**)** returns $\pi/2 - i\infty$, for 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(**$\pm\infty + i\mathrm{NaN}$**)** returns $\mathrm{NaN} \pm i\infty$ (where the sign of the imaginary part of the result is unspecified).

— **cacos(**$\pm 0 + i\mathrm{NaN}$**)** returns $\pi/2 + i\mathrm{NaN}$.

— **cacos(**$\mathrm{NaN} + i\infty$**)** returns $\mathrm{NaN} - i\infty$.

— **cacos(**$x + i\mathrm{NaN}$**)** returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the *invalid* exception, for nonzero finite x.

— **`cacos(`**$\mathrm{NaN} + iy$**`)`** returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the *invalid* exception, for finite y.

— **`cacos(`**$\mathrm{NaN} + i\mathrm{NaN}$**`)`** returns $\mathrm{NaN} + i\mathrm{NaN}$.

G.5.2 Hyperbolic functions

G.5.2.1 The `cacosh` functions

1 — **`cacosh(conj(`**z**`)) = conj(cacosh(`**z**`))`**.

— **`cacosh(`**$\pm 0 + i0$**`)`** returns $+0 + i\pi/2$.

— **`cacosh(`**$-\infty + i\infty$**`)`** returns $+\infty + i3\pi/4$.

— **`cacosh(`**$+\infty + i\infty$**`)`** returns $+\infty + i\pi/4$.

— **`cacosh(`**$x + i\infty$**`)`** returns $+\infty + i\pi/2$, 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(`**$\mathrm{NaN} + i\infty$**`)`** returns $+\infty + i\mathrm{NaN}$.

— **`cacosh(`**$\pm\infty + i\mathrm{NaN}$**`)`** returns $+\infty + i\mathrm{NaN}$.

— **`cacosh(`**$x + i\mathrm{NaN}$**`)`** returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the *invalid* exception, for finite x.

— **`cacosh(`**$\mathrm{NaN} + iy$**`)`** returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the *invalid* exception, for finite y.

— **`cacosh(`**$\mathrm{NaN} + i\mathrm{NaN}$**`)`** returns $\mathrm{NaN} + i\mathrm{NaN}$.

G.5.2.2 The `casinh` functions

— **`casinh(conj(`**z**`)) = conj(casinh(`**z**`))`** and **`casinh`** is odd.

— **`casinh(`**$+0 + i0$**`)`** returns $0 + i0$.

— **`casinh(`**$\infty + i\infty$**`)`** returns $+\infty + i\pi/4$.

— **`casinh(`**$x + i\infty$**`)`** returns $+\infty + i\pi/2$ for positive-signed finite x.

— **`casinh(`**$+\infty + iy$**`)`** returns $+\infty + i0$ for positive-signed finite y.

— **`casinh(`**$\mathrm{NaN} + i\infty$**`)`** returns $\pm\infty + i\mathrm{NaN}$ (where the sign of the real part of the result is unspecified).

— **`casinh(`**$+\infty + i\mathrm{NaN}$**`)`** returns $+\infty + i\mathrm{NaN}$.

— **`casinh(`**$\mathrm{NaN} + i0$**`)`** returns $\mathrm{NaN} + i0$.

— **`casinh(`**$\mathrm{NaN} + iy$**`)`** returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the *invalid* exception, for finite nonzero y.

— **`casinh(`**$x + i\mathrm{NaN}$**`)`** returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the *invalid* exception, for finite x.

— **casinh(**NaN + *i*NaN**)** returns NaN + *i*NaN.

G.5.2.3 The catanh functions

1 — **catanh(conj(***z***)) = conj(catanh(***z***))** and **catanh** is odd.

— **catanh(**+0 + *i*0**)** returns +0 + *i*0.

— **catanh(**+∞ + *i*∞**)** returns +0 + $i\pi/2$.

— **catanh(**+∞ + *iy***)** returns +0 + $i\pi/2$, for finite positive-signed *y*.

— **catanh(***x* + *i*∞**)** returns +0 + $i\pi/2$, for finite positive-signed *x*.

— **catanh(**+0 + *i*NaN**)** returns +0 + *i*NaN.

— **catanh(**NaN + *i*∞**)** returns ±0 + $i\pi/2$ (where the sign of the real part of the result is unspecified).

— **catanh(**+∞ + *i*NaN**)** returns +0 + *i*NaN.

— **catanh(**NaN + *iy***)** returns NaN + *i*NaN and optionally raises the *invalid* exception, for finite *y*.

— **catanh(***x* + *i*NaN**)** returns NaN + *i*NaN and optionally raises the *invalid* exception, for nonzero finite *x*.

— **catanh(**NaN + *i*NaN**)** returns NaN + *i*NaN.

G.5.2.4 The ccosh functions

1 — **ccosh(conj(***z***)) = conj(ccosh(***z***))** and **ccosh** is even.

— **ccosh(**+0 + *i*0**)** returns 1 + *i*0.

— **ccosh(**+0 + *i*∞**)** returns NaN ± *i*0 (where the sign of the imaginary part of the result is unspecified) and raises the *invalid* exception.

— **ccosh(**+∞ + *i*0**)** returns +∞ + *i*0.

— **ccosh(**+∞ + *i*∞**)** returns +∞ + *i*NaN and raises the *invalid* exception.

— **ccosh(***x* + *i*∞**)** returns NaN + *i*NaN and raises the *invalid* exception, for finite nonzero *x*.

— **ccosh(**+∞ + *iy***)** returns +∞ cis(*y*), for finite nonzero *y*.

— **ccosh(**+0 + *i*NaN**)** returns NaN ± *i*0 (where the sign of the imaginary part of the result is unspecified).

— **ccosh(**+∞ + *i*NaN**)** returns +∞ + *i*NaN.

— **ccosh(***x* + *i*NaN**)** returns NaN + *i*NaN and optionally raises the *invalid* exception, for finite nonzero *x*.

— **ccosh(**NaN + *i*0**)** returns NaN ± *i*0 (where the sign of the imaginary part of the result is unspecified).

— **ccosh(**NaN + iy**)** returns NaN + iNaN and optionally raises the *invalid* exception, for all nonzero numbers y.

— **ccosh(**NaN + iNaN**)** returns NaN + iNaN.

G.5.2.5 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 ±0 + iNaN (where the sign of the real part of the result is unspecified) and raises the *invalid* exception.

— **csinh(**+∞ + i0**)** returns +∞ + i0.

— **csinh(**+∞ + $i\infty$**)** returns ±∞ + iNaN (where the sign of the real part of the result is unspecified) and raises the *invalid* exception.

— **csinh(**+∞ + iy**)** returns +∞ cis(y), for positive finite y.

— **csinh(**$x + i\infty$**)** returns NaN + iNaN and raises the *invalid* exception, for positive finite x.

— **csinh(**+0 + iNaN**)** returns ±0 + iNaN (where the sign of the real part of the result is unspecified).

— **csinh(**+∞ + iNaN**)** returns ±∞ + iNaN (where the sign of the real part of the result is unspecified).

— **csinh(**x + iNaN**)** returns NaN + iNaN and optionally raises the *invalid* exception, for finite nonzero x.

— **csinh(**NaN + i0**)** returns NaN + i0.

— **csinh(**NaN + iy**)** returns NaN + iNaN and optionally raises the *invalid* exception, for all nonzero numbers y.

— **csinh(**NaN + iNaN**)** returns NaN + iNaN.

G.5.2.6 The `ctanh` functions

1 — `ctanh(conj(`z`)) = conj(ctanh(`z`))` and `ctanh` is odd.

— `ctanh(`$+0 + i0$`)` returns $+0 + i0$.

— `ctanh(`$+\infty + iy$`)` returns $1 + i0$, for all positive-signed numbers y.

— `ctanh(`$x + i\infty$`)` returns NaN + iNaN and raises the *invalid* exception, for finite x.

— `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* exception, for all nonzero numbers y.

— `ctanh(`$x + i$NaN`)` returns NaN + iNaN and optionally raises the *invalid* exception, for finite x.

— `ctanh(`NaN + iNaN`)` returns NaN + iNaN.

G.5.3 Exponential and logarithmic functions

G.5.3.1 The `cexp` functions

1 — `cexp(conj(`z`)) = conj(cexp(`z`))`.

— `cexp(`$\pm 0 + i0$`)` returns $1 + i0$.

— `cexp(`$+\infty + i0$`)` returns $+\infty + i0$.

— `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* exception (where the sign of the real part of the result is unspecified).

— `cexp(`$x + i\infty$`)` returns NaN + iNaN and raises the *invalid* exception, for finite x.

— `cexp(`$-\infty + iy$`)` returns $+0$ cis(y), for finite y.

— `cexp(`$+\infty + iy$`)` returns $+\infty$ cis(y), for finite nonzero y.

— `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 + iNaN and optionally raises the *invalid* exception, for all non-zero numbers y.

— `cexp(`$x + i$NaN`)` returns NaN + iNaN and optionally raises the *invalid* exception, for finite x.

— **`cexp(`**NaN + iNaN**`)`** returns NaN + iNaN.

G.5.3.2 The `clog` functions

1 — **`clog(conj(`**z**`)) = conj(clog(`**z**`))`**.

— **`clog(`**$-0 + i0$**`)`** returns $-\infty + i\pi$ and raises the *divide-by-zero* exception.

— **`clog(`**$+0 + i0$**`)`** returns $-\infty + i0$ and raises the *divide-by-zero* exception.

— **`clog(`**$-\infty + i\infty$**`)`** returns $+\infty + i3\pi/4$.

— **`clog(`**$+\infty + i\infty$**`)`** returns $+\infty + i\pi/4$.

— **`clog(`**$x + i\infty$**`)`** returns $+\infty + i\pi/2$, 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(`**$\pm\infty$ + iNaN**`)`** returns $+\infty$ + iNaN.

— **`clog(`**NaN + $i\infty$**`)`** returns $+\infty$ + iNaN.

— **`clog(`**x + iNaN**`)`** returns NaN + iNaN and optionally raises the *invalid* exception, for finite x.

— **`clog(`**NaN + iy**`)`** returns NaN + iNaN and optionally raises the *invalid* exception, for finite y.

— **`clog(`**NaN + iNaN**`)`** returns NaN + iNaN.

G.5.4 Power and absolute-value functions

G.5.4.1 The `csqrt` functions

1 — **`csqrt(conj(`**z**`)) = conj(csqrt(`**z**`))`**.

— **`csqrt(`**$\pm 0 + i0$**`)`** returns $+0 + i0$.

— **`csqrt(`**$-\infty + iy$**`)`** returns $+0 + i\infty$, for finite positive-signed y.

— **`csqrt(`**$+\infty + iy$**`)`** returns $+\infty + i0$, for finite positive-signed y.

— **`csqrt(`**$x + i\infty$**`)`** returns $+\infty + i\infty$, for all x (including NaN).

— **`csqrt(`**$-\infty$ + iNaN**`)`** returns NaN $\pm\, i\infty$ (where the sign of the imaginary part of the result is unspecified).

— **`csqrt(`**$+\infty$ + iNaN**`)`** returns $+\infty$ + iNaN.

— **`csqrt(`**x + iNaN**`)`** returns NaN + iNaN and optionally raises the *invalid* exception, for finite x.

— **`csqrt(`**NaN + iy**`)`** returns NaN + iNaN and optionally raises the *invalid* exception, for finite y.

— **`csqrt(`**NaN + iNaN**`)`** returns NaN + iNaN.

G.6 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:

cos(iy**)** = **cosh(**y**)**
sin(iy**)** = i **sinh(**y**)**
tan(iy**)** = i **tanh(**y**)**
cosh(iy**)** = **cos(**y**)**
sinh(iy**)** = i **sin(**y**)**
tanh(iy**)** = i **tan(**y**)**
asin(iy**)** = i **asinh(**y**)**
atan(iy**)** = i **atanh(**y**)**
asinh(iy**)** = i **asin(**y**)**
atanh(iy**)** = i **atan(**y**)**

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. User-provided signal handlers for **SIGFPE** allow for trap-and-resume behavior.

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. 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 arithmetic exceptions, if there is any signal raised for them.

4 C supports signal handlers for **SIGFPE** and allows trapping of arithmetic exceptions. When 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
(normative)

Universal character names for identifiers

1 This clause lists the hexadecimal code values that are valid in universal character names in identifiers.

2 This table is reproduced unchanged from ISO/IEC TR 10176, produced by ISO/IEC JTC1/SC22/WG20, except for the omission of ranges that are part of the required basic source character set.

Latin:	00AA, 00BA, 00C0−00D6, 00D8−00F6, 00F8−01F5, 01FA−0217, 0250−02A8, 1E00−1E9B, 1EA0−1EF9, 207F
Greek:	0386, 0388−038A, 038C, 038E−03A1, 03A3−03CE, 03D0−03D6, 03DA, 03DC, 03DE, 03E0, 03E2−03F3, 1F00−1F15, 1F18−1F1D, 1F20−1F45, 1F48−1F4D, 1F50−1F57, 1F59, 1F5B, 1F5D, 1F5F−1F7D, 1F80−1FB4, 1FB6−1FBC, 1FC2−1FC4, 1FC6−1FCC, 1FD0−1FD3, 1FD6−1FDB, 1FE0−1FEC, 1FF2−1FF4, 1FF6−1FFC
Cyrillic:	0401−040C, 040E−044F, 0451−045C, 045E−0481, 0490−04C4, 04C7−04C8, 04CB−04CC, 04D0−04EB, 04EE−04F5, 04F8−04F9
Armenian:	0531−0556, 0561−0587
Hebrew:	05B0−05B9, 05BB−05BD, 05BF, 05C1−05C2, 05D0−05EA, 05F0−05F2
Arabic:	0621−063A, 0640−0652, 0670−06B7, 06BA−06BE, 06C0−06CE, 06D0−06DC, 06E5−06E8, 06EA−06ED
Devanagari:	0901−0903, 0905−0939, 093E−094D, 0950−0952, 0958−0963
Bengali:	0981−0983, 0985−098C, 098F−0990, 0993−09A8, 09AA−09B0, 09B2, 09B6−09B9, 09BE−09C4, 09C7−09C8, 09CB−09CD, 09DC−09DD, 09DF−09E3, 09F0−09F1
Gurmukhi:	0A02, 0A05−0A0A, 0A0F−0A10, 0A13−0A28, 0A2A−0A30, 0A32−0A33, 0A35−0A36, 0A38−0A39, 0A3E−0A42, 0A47−0A48, 0A4B−0A4D, 0A59−0A5C, 0A5E, 0A74
Gujarati:	0A81−0A83, 0A85−0A8B, 0A8D, 0A8F−0A91, 0A93−0AA8, 0AAA−0AB0, 0AB2−0AB3, 0AB5−0AB9, 0ABD−0AC5, 0AC7−0AC9, 0ACB−0ACD, 0AD0, 0AE0
Oriya:	0B01−0B03, 0B05−0B0C, 0B0F−0B10, 0B13−0B28, 0B2A−0B30, 0B32−0B33, 0B36−0B39, 0B3E−0B43, 0B47−0B48, 0B4B−0B4D, 0B5C−0B5D, 0B5F−0B61

Tamil:	0B82−0B83, 0B85−0B8A, 0B8E−0B90, 0B92−0B95, 0B99−0B9A, 0B9C, 0B9E−0B9F, 0BA3−0BA4, 0BA8−0BAA, 0BAE−0BB5, 0BB7−0BB9, 0BBE−0BC2, 0BC6−0BC8, 0BCA−0BCD
Telugu:	0C01−0C03, 0C05−0C0C, 0C0E−0C10, 0C12−0C28, 0C2A−0C33, 0C35−0C39, 0C3E−0C44, 0C46−0C48, 0C4A−0C4D, 0C60−0C61
Kannada:	0C82−0C83, 0C85−0C8C, 0C8E−0C90, 0C92−0CA8, 0CAA−0CB3, 0CB5−0CB9, 0CBE−0CC4, 0CC6−0CC8, 0CCA−0CCD, 0CDE, 0CE0−0CE1
Malayalam:	0D02−0D03, 0D05−0D0C, 0D0E−0D10, 0D12−0D28, 0D2A−0D39, 0D3E−0D43, 0D46−0D48, 0D4A−0D4D, 0D60−0D61
Thai:	0E01−0E3A, 0E40−0E5B
Lao:	0E81−0E82, 0E84, 0E87−0E88, 0E8A, 0E8D, 0E94−0E97, 0E99−0E9F, 0EA1−0EA3, 0EA5, 0EA7, 0EAA−0EAB, 0EAD−0EAE, 0EB0−0EB9, 0EBB−0EBD, 0EC0−0EC4, 0EC6, 0EC8−0ECD, 0EDC−0EDD
Tibetan:	0F00, 0F18−0F19, 0F35, 0F37, 0F39, 0F3E−0F47, 0F49−0F69, 0F71−0F84, 0F86−0F8B, 0F90−0F95, 0F97, 0F99−0FAD, 0FB1−0FB7, 0FB9
Georgian:	10A0−10C5, 10D0−10F6
Hiragana:	3041−3093, 309B−309C
Katakana:	30A1−30F6, 30FB−30FC
Bopomofo:	3105−312C
CJK Unified Ideographs:	4E00−9FA5
Hangul:	AC00−D7A3
Digits:	0660−0669, 06F0−06F9, 0966−096F, 09E6−09EF, 0A66−0A6F, 0AE6−0AEF, 0B66−0B6F, 0BE7−0BEF, 0C66−0C6F, 0CE6−0CEF, 0D66−0D6F, 0E50−0E59, 0ED0−0ED9, 0F20−0F33
Special characters:	00B5, 00B7, 02B0−02B8, 02BB, 02BD−02C1, 02D0−02D1, 02E0−02E4, 037A, 0559, 093D, 0B3D, 1FBE, 203F−2040, 2102, 2107, 210A−2113, 2115, 2118−211D, 2124, 2126, 2128, 212A−2131, 2133−2138, 2160−2182, 3005−3007, 3021−3029

Annex J
(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).

— 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 enumeration type other than by assignment of an enumeration constant that is a member of that type, or an enumeration variable that has the same type, or the value of a function that returns the same enumeration type (6.7.2.2).

— An aggregate has a partly bracketed initialization (6.7.7).

— 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 K
(informative)

Portability issues

1 This annex collects some information about portability that appears in this International Standard.

K.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 behavior if a printable character is written when the active position is at the final position of a line (5.2.2).

— The behavior if a backspace character is written when the active position is at the initial position of a line (5.2.2).

— The behavior 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 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 value of a union member other than the last one 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 value of padding bits in integer representations (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).

— The choice of using an inline definition or external definition of a function when both definitions are in scope (6.7.4).

— The size of an array when `*` is written instead of a size expression (6.7.5.2).

— Whether side effects are produced when evaluating the size expression in a declaration of a variable length array (6.7.5.2).

— 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).

— 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 order in which `#` and `##` operations are evaluated during macro substitution (6.10.3.2, 6.10.3.3).

— Whether **`errno`** is a macro or an external identifier (7.5).

— The order of raising floating-point exceptions, except as stated in F.7.6 (7.6.2.3).

— The result of rounding when the value is out of range (7.12.9.5, 7.12.9.7, F.9.6.5).

— Whether **`setjmp`** is a macro or an external identifier (7.13).

— Whether **`va_end`** is a macro or an external identifier (7.15.1).

— The hexadecimal digit left of the decimal point when a non-normalized floating-point number is printed with an **`a`** or **`A`** conversion specifier (7.19.6.1, 7.24.2.1).

— 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 (7.19.7.11, 7.24.3.10).

— The details of the value stored by the **`fgetpos`** function (7.19.9.1).

— The details of the value returned by the **`ftell`** function for a text stream (7.19.9.4).

— The order and contiguity of storage allocated by successive calls to the **`calloc`**, **`malloc`**, and **`realloc`** functions (7.20.3).

— The amount of storage allocated by a successful call to the **`calloc`**, **`malloc`**, or **`realloc`** function when 0 bytes was requested (7.20.3).

— Which of two elements that compare as equal is matched by the **`bsearch`** function (7.20.5.1).

— The order of two elements that compare as equal in an array sorted by the **qsort** function (7.20.5.2).

— The value of the **tm_extlen** member of the **tmx** structure to which a pointer is returned by the **zonetime** function when it has set the **tm_ext** member to a null pointer (7.23.1).

— The encoding of the calendar time returned by the **time** function (7.23.2.5).

— The resulting value when the invalid exception is raised during IEC 60559 floating to integer conversion (F.4).

— Whether conversion of non-integer IEC 60559 floating values raises the inexact exception (F.4).

— The value stored by **frexp** for a NaN or infinity (F.9.3.4).

— The sign of one part of the **complex** result of several math functions for certain exceptional values in IEC 60559 compatible implementations (G.5.1.1, G.5.2.2, G.5.2.3, G.5.2.4, G.5.2.5, G.5.2.6, G.5.3.1, G.5.4.1).

K.2 Undefined behavior

1 The behavior is undefined in the following circumstances:

— A nonempty source file does not end in a new-line character, ends in new-line character immediately preceded by a backslash character, or ends in a partial preprocessing token or comment (5.1.1.2).

— Physical source line splicing or token concatenation produces a character sequence matching the syntax of a universal character name (5.1.1.2).

— A universal character name specifies a character identifier in the range 00000000 through 00000020 or 0000007F through 0000009F, or designates a character in the required source character set (5.1.1.2).

— A program in a hosted environment does not define a function named **main** using one of the two specified forms (5.1.2.2.1).

— A character not in the required 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 is used more than once as a label in the same function (6.2.1).

— The same identifier has both internal and external linkage in the same translation unit (6.2.2).

— The value of an object with automatic storage duration is used while it is indeterminate. (6.2.4, 6.7.8, 6.8.2).

- An object is referred to when storage is not allocated for it (6.2.4).
- The value of a pointer that referred to an object with automatic storage duration is used after the storage is no longer guaranteed to be reserved (6.2.4).
- A trap representation is accessed 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).
- Two declarations of the same object or function specify types that are not compatible (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 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 to a function is converted to point to a function of a different type and used to call a function of a type not compatible with the type of the called function (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 does not designate a code value in one of the specified ranges (6.4.2.1).
- The first character of an identifier is a universal character name designating an extended digit (6.4.2.1).

— Two identifiers differ only in nonsignificant characters (6.4.2.1).

— An unspecified escape sequence is encountered in a character constant or a string literal (6.4.4.4).

— An attempt is made to modify a string literal of either form (6.4.5).

— The characters `'`, `\`, `"`, `//`, or `/*` are encountered between the `<` and `>` delimiters, or the characters `'`, `\`, `//`, or `/*` are encountered between the `"` delimiters, in a header name preprocessing token (6.4.7).

— Between two sequence points, an object is modified more than once, or is modified and the prior value is accessed other than to determine the value to be stored (6.5).

— An exception occurs during the evaluation of an expression (6.5).

— An object has its stored value accessed other than by an lvalue expression having one of the following types: 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 (6.5).

— For a function call without a function prototype, the number of arguments does not agree with the number of parameters (6.5.2.2).

— For a function call without a function prototype, the function is defined without a function prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, with certain exceptions (6.5.2.2).

— For a function call without a function prototype, the function is defined with a function prototype, and the types of the arguments after promotion are not compatible with the types of the parameters, or the prototype ends with an ellipsis (6.5.2.2).

— A function is defined with a type that is not compatible with the type pointed to by the expression that denotes the called function (6.5.2.2).

— An attempt is made to modify the result of a function call or to access it after the next sequence point (6.5.2.2).

— 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 attempt is made to modify the result of a conditional operator or to access it after the next sequence point (6.5.15).

— An attempt is made to modify the result of an assignment operator or to access it after the next sequence point (6.5.16).

— An object is assigned to an inexactly overlapping object or to an exactly overlapping object with incompatible type (6.5.16.1).

— An attempt is made to modify the result of a comma operator or to access it after the next sequence point (6.5.17).

— An expression that is required to be an integer constant expression does not have an integer type, contains casts (outside operands to `sizeof` operators) other than conversions of arithmetic types to integer types, or has operands that are not integer constants, enumeration constants, character constants, fixed-length `sizeof` expressions, or immediately-cast floating constants (6.6).

— A constant expression in an initializer does not evaluate to one of the following: an arithmetic constant expression, a null pointer constant, an address constant, or an address constant for an object type plus or minus an integer constant expression (6.6).

— An arithmetic constant expression does not have arithmetic type, contains casts (outside operands to `sizeof` operators) other than conversions of arithmetic types to arithmetic types, or has operands that are not integer constants, floating constants, enumeration constants, character constants, or `sizeof` expressions (6.6).

— An address constant is created neither explicitly using the unary **&** operator or an integer constant cast to pointer type, nor implicitly by the use of an expression of array or function type (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 as containing no named members (6.7.2.1).

— A bit-field is declared with a type other than a qualified or unqualified version of **_Bool**, **signed int**, or **unsigned int** (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).

— A tag is declared with the bracketed list twice within the same scope (6.7.2.3).

— 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).

— An attempt is made to access an object through a restrict-qualified pointer and another pointer not based on it (6.7.3, 6.7.3.1).

— The specification of a function type includes a type qualifier (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).

— 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).

— Two pointer types that are required to be compatible are not identically qualified, or are not pointers to compatible types (6.7.5.1).

— The size expression in an array declaration is not a constant expression and evaluates at program execution time to a nonpositive value (6.7.5.2).

— 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 (6.7.5.2).

— A storage-class specifier or type qualifier modifies the keyword **`void`** as a function parameter type list (6.7.5.3).

— 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 identifier list) (6.7.5.3).

— The value of an unnamed member of a structure or union is used (6.7.8). *

— The initializer for a scalar is neither a single expression nor a single expression enclosed in braces (6.7.8). *

— The initializer for a structure or union object is neither an initializer list nor a single expression that has compatible structure or union type (6.7.8).

— 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. (6.7.8). |

— 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).

— 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).

— An adjusted parameter type in a function definition is not an object type (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 directive lines (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 nine well-defined forms (6.10.6).

— One of the following identifiers is the subject of a **#define** or **#undef** preprocessing directive: **__LINE__**, **__FILE__**, **__DATE__**, **__TIME__**, **__STDC__**, **__STDC_VERSION__**, **__STDC_IEC_559__**, **__STDC_IEC_559_COMPLEX__**, or **defined** (6.10.8).

— An attempt is made to copy an object to an overlapping object by use of a library function other than **memmove** (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 macro definition of **assert** is suppressed in order to access an actual function (7.2).

— The **CX_LIMITED_RANGE** 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).

— 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).

— The **FENV_ACCESS** pragma is used in any context other than outside all external declarations or preceding all explicit declarations and statements inside a compound statement (7.6.1).

— Part of the program tests flags 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 exception flags has a value not obtained by bitwise OR of the exception macros (7.6.2).

— The **fesetexceptflag** function is used to set the status for exception flags not specified in the call to the **fegetexceptflag** function that provided the value of the corresponding **fexcept_t** object (7.6.2.4).

— 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).

— The **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 (7.12.2).

— An argument to a floating-point classification macro is not of real floating type (7.12.3).

— A macro definition of **setjmp** is suppressed in order to access an actual function, or the program defines an identifier with the name **setjmp** (7.13.1).

— An invocation of the **setjmp** macro occurs in a context other than as the entire controlling expression in a selection or iteration statement, or in a comparison with an integer constant expression (possibly as implied by the unary **!** operator) as the entire controlling expression of a selection or iteration statement, or as the entire expression of an expression statement (possibly cast to **void**) (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 class with non-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 **longjmp** function is invoked from a nested signal handler (7.13.2.1).

— The program uses a nonpositive value for a signal number (7.14).

— 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 occurs other than as the result of calling the **abort** or **raise** function, and the signal handler calls a function in the standard library other than the **signal** function (for the same signal number) or refers to an object with static storage duration other than by assigning a value to an object declared as **volatile sig_atomic_t** (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 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 (7.15, 7.15.1.1, 7.15.1.4).

— 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 (7.15).

— 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_end** (7.15.1).

— The **va_end** macro is invoked without a corresponding invocation of the **va_start** or **va_copy** macro, or vice versa. (7.15.1, 7.15.1.3, 7.15.1.4).

— 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 (7.15.1.1).

— 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 (7.15.1.4).

— The *member-designator* parameter of an **offsetof** macro is an invalid right operand of the **.** operator for the *type* parameter, or the *member-designator* parameter designates a bit-field (7.17).

— 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 (7.18.4).

— 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 (7.19.2).

— Use is made of any portion of a file beyond the most recent wide character written to a wide-oriented stream (7.19.2).

— The value of a pointer to a **FILE** object is used after the associated file is closed (7.19.3).

— The stream for the **fflush** function points to an input stream or to an update stream in which the most recent operation was input (7.19.5.2).

— 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 (7.19.5.3).

— 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 (7.19.5.3).

— An attempt is made to use the contents of the array that was supplied in a call to the **setvbuf** function (7.19.5.6).

— There are insufficient arguments for the format in a call to the **fprintf**, **fscanf**, **fwprintf**, or **fwscanf** function, or an argument does not have an appropriate type (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2).

— The format in a call to the **fprintf**, **fscanf**, **fwprintf**, **fwscanf**, **strftime**, **strfxtime**, **wcsftime**, or **wcsfxtime** function is not a valid multibyte character sequence that begins and ends in its initial shift state (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2, 7.23.3.5, 7.23.3.6, 7.24.5.1, 7.24.5.2).

— In a call to the **fprintf** or **fwprintf** function, a precision appears with a conversion specifier other than **a**, **A**, **d**, **e**, **E**, **f**, **F**, **g**, **G**, **i**, **o**, **s**, **u**, **x**, or **X** (7.19.6.1, 7.24.2.1).

— A conversion specification for the **fprintf**, **fscanf**, **fwprintf**, or **fwscanf** function uses a length modifier with a conversion specifier in a combination not specified in this International Standard (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2).

— An asterisk is used to denote an argument-supplied field width or precision, but the corresponding argument is not provided (7.19.6.1, 7.24.2.1).

— A conversion specification for the **fprintf** or **fwprintf** function uses a **#** flag with a conversion specifier other than **a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**, **o**, **x**, or **X** (7.19.6.1, 7.24.2.1).

— A conversion specification for the **fprintf** or **fwprintf** function uses a **0** flag with a conversion specifier other than **a**, **A**, **d**, **e**, **E**, **f**, **F**, **g**, **G**, **i**, **o**, **u**, **x**, or **X** (7.19.6.1, 7.24.2.1).

— An **s** conversion specifier is encountered by the **fprintf** or **fwprintf** function, and the argument is missing the null terminator (unless a precision is specified that does not require null termination) (7.19.6.1, 7.24.2.1).

— An **n** conversion specification includes any flags, an assignment-suppressing character, a field width, or a precision (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2).

— A **%** conversion specifier is encountered by the **fprintf**, **fscanf**, **fwprintf**, or **fwscanf** function, but the complete conversion specification is not exactly **%%** (7.19.6.1, 7.19.6.2, 7.24.2.1, 7.24.2.2).

— An invalid conversion specification is found in the format for the **fprintf**, **fscanf**, **strftime**, **strfxtime**, **fwprintf**, **fwscanf**, **wcsftime**, or **wcsfxtime** function (7.19.6.1, 7.19.6.2, 7.23.3.5, 7.23.3.6, 7.24.2.1, 7.24.2.2, 7.24.5.1, 7.24.5.2).

— An argument to the **fprintf** or **fwprintf** function is not the correct type for the corresponding conversion specification (7.19.6.1, 7.24.2.1).

— The number of characters transmitted by a formatted output function is greater than **INT_MAX** (7.19.6.1, 7.19.6.3, 7.19.6.8, 7.19.6.10)

— The result of a conversion by the **fscanf** or **fwscanf** function cannot be represented in the corresponding object, or the receiving object does not have an appropriate type (7.19.6.2, 7.24.2.2).

— A **c**, **s**, or **[** conversion specifier is encountered by the **fscanf** or **fwscanf** function, and the character 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 **[**) (7.19.6.2, 7.24.2.2).

— An **c**, **s**, or **[** conversion specifier with an **l** qualifier is encountered by the **fscanf** or **fwscanf** function, but the input is not a valid multibyte character sequence that begins in the initial shift state (7.19.6.2, 7.24.2.2).

— The input item for **%p** conversion by the **fscanf** or **fwscanf** function is not a value converted earlier during the same program execution (7.19.6.2, 7.24.2.2).

— The **snprintf**, **sprintf**, **sscanf**, **vsnprintf**, **vsprintf**, **mbstowcs**, **wcstombs**, **memcpy**, **strcpy**, **strncpy**, **strcat**, **strncat**, **strxfrm**, **strftime**, or **strfxtime** function, or any of the functions declared by **<wchar.h>** (except where otherwise specified), is used to copy between overlapping objects (7.19.6.5, 7.19.6.6, 7.19.6.7, 7.19.6.12, 7.19.6.13, 7.20.8.1, 7.20.8.2, 7.21.2.1, 7.21.2.3, 7.21.2.4, 7.21.3.1, 7.21.3.2, 7.21.4.5, 7.23.3.5, 7.23.3.6, 7.24.1).

— 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 (7.19.6.8, 7.19.6.9, 7.19.6.10, 7.19.6.11, 7.19.6.12, 7.19.6.13, 7.19.6.14, 7.24.2.5, 7.24.2.6, 7.24.2.7, 7.24.2.8, 7.24.2.9, 7.24.2.10).

— The contents of the array supplied in a call to the **fgets**, **gets**, or **fgetws** function are used after a read error occurred (7.19.7.2, 7.19.7.7, 7.24.3.2).

— 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 (7.19.7.11).

— The file position indicator for a stream is used after an error occurred during a call to the **fread** or **fwrite** function (7.19.8.1, 7.19.8.2).

— A partial element read by a call to the **fread** function is used (7.19.8.1).

— The **fseek** function is called for a text stream with other than **SEEK_SET**, or with a non-zero offset that was not returned by a previous successful call to the **ftell** function for the same file (7.19.9.2).

— The **fsetpos** function is called to set a position that was not returned by a previous successful call to the **fgetpos** function for the same file (7.19.9.3).

— The value of the result of converting a string to a number by the **atof**, **atoi**, **atol**, or **atoll** function cannot be represented (7.20.1).

— 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 (7.20.3).

— The value of a pointer that refers to space deallocated by a call to the **free** or **realloc** function is used (7.20.3).

— The pointer argument to the **free** or **realloc** function does not match a pointer earlier returned by **calloc**, **malloc**, or **realloc**, or the space has been deallocated by a call to **free** or **realloc** (7.20.3.2, 7.20.3.4).

— The value of the object allocated by the **malloc** function is used (7.20.3.3).

— The value of the newly allocated portion of an object expanded by the **realloc** function is used (7.20.3.4).

— The program executes more than one call to the **exit** function (7.20.4.3).

— The string set up by the **getenv** or **strerror** function is modified by the program (7.20.4.4, 7.21.6.2).

— A command is executed through the **system** function in a way that is documented as causing termination or some other form of undefined behavior (7.20.4.5).

— The comparison function called by the **bsearch** or **qsort** function returns ordering values inconsistently (7.20.5.1, 7.20.5.2).

— The array being searched by the **bsearch** function does not have its elements in proper order (7.20.5.1).

— The result of an integer arithmetic function (**abs**, **div**, **labs**, **llabs**, **ldiv**, or **lldiv**) cannot be represented (7.20.6.1, 7.20.6.2).

— The current shift state is used with a multibyte character function after the **LC_CTYPE** category was changed (7.20.7).

— A string or wide-string utility function is instructed to access an array beyond the end of an object (7.21.1, 7.24.4).

— The contents of the destination array are used after a call to the **strxfrm**, **strftime**, **strfxtime**, **wcsxfrm**, **wcsftime**, or **wcsfxtime** function in which the specified length was too small to hold the entire null-terminated result (7.21.4.5, 7.23.3.5, 7.23.3.6, 7.24.4.4.4, 7.24.5.1, 7.24.5.2).

— A non-real argument is supplied for a generic parameter of a type-generic macro (7.22.1).

— The **tmx** structure whose address is passed as an argument to the **mkxtime**, **strfxtime**, or **wcsfxtime** function does not provide an appropriate extension block as required by the implementation (7.23.1).

— The value of the **tm_version** member of the **tmx** structure pointed to by the argument in a call to the **mkxtime** function is not 1 (7.23.2.3, 7.23.2.6).

— The value of one of the **tm_year**, **tm_mon**, **tm_mday**, **tm_hour**, **tm_min**, **tm_sec**, **tm_leapsecs**, **tm_zone**, or **tm_isdst** members of the **tm** or **tmx** structure pointed to by the argument in a call to the **mktime** or **mkxtime** function is drastically out of the normal range (7.23.2.6).

— 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 (7.24.2.11).

— The first argument in a call to the **wcstok** function does not point to a wide string on the first call or is not a null pointer for subsequent calls to continue parsing the same wide string, or when continuing parsing, the saved pointer value pointed to by the third argument does not match that stored by the previous call for the same wide string (7.24.4.5.8).

— An **mbstate_t** object is used inappropriately (7.24.6).

— The conversion state is used after the **mbrtowc**, **wcrtomb**, **mbsrtowcs**, or **wcsrtombs** function reports an encoding error (7.24.6.3.2, 7.24.6.3.3, 7.24.6.4.1, 7.24.6.4.2).

— The value of an argument of type **wint_t** to a wide-character classification or mapping function is neither equal to the value of **WEOF** nor representable as a **wchar_t** (7.25.1).

— 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 (7.25.2.2.1).

— 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 (7.25.3.2.1).

K.3 Implementation-defined behavior

1 A conforming implementation shall document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:

K.3.1 Translation

1 — How a diagnostic is identified (3.8, 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).

K.3.2 Environment

1 — 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).

— Signals for which the equivalent of **`signal(sig, SIG_IGN);`** is executed at program startup (7.14.1.1).

— The form of the status returned to the host environment to indicate unsuccessful termination when the **`SIGABRT`** signal is raised and not caught (7.20.4.1).

— The forms of the status returned to the host environment by the **`exit`** function to report successful and unsuccessful termination (7.20.4.3).

— The status returned to the host environment by the **`exit`** function if the value of its argument is other than zero, **`EXIT_SUCCESS`**, or **`EXIT_FAILURE`** (7.20.4.3).

— The set of environment names and the method for altering the environment list used by the **`getenv`** function (7.20.4.4).

— The manner of execution of the string by the **`system`** function (7.20.4.5).

K.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).

K.3.4 Characters

1 — The number of bits in a byte (3.4).

— 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 required source 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).

— The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (6.4.4.4).

— The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or a wide character constant that contains 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).

— 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).

K.3.5 Integers

1 — Any extended integer types that exist in the implementation (6.2.5).

— The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

— The result of 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 bit-wise operations on signed integers (6.5).

K.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)

— Rounding behavior for values of **FLT_ROUNDS** less than −1 (5.2.4.2.2).

— Rounding behavior for values of **FLT_EVAL_METHOD** less than −1 or greater than 2 (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).

K.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).

K.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).

K.3.9 Structures, unions, enumerations, and bit-fields

1 — The behavior when a member of a union object is accessed using a member of a different type (6.5.2.3).

— Whether a "plain" **int** bit-field is treated as a **signed int** bit-field or as an **unsigned int** bit-field (6.7.2).

— 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).

K.3.10 Qualifiers

1 — What constitutes an access to an object that has volatile-qualified type (6.7.3).

K.3.11 Preprocessing directives

1 — 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 are combined into a header name (6.10.2).

— The nesting limit for `#include` processing (6.10.2).

— 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).

K.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 default state for the `FENV_ACCESS` pragma (7.6.1)

— The representation of floating point exception flags stored by the `fegetexceptflag` function (7.6.2.2).

— Whether the `feraiseexcept` function raises the inexact exception in addition to the overflow or underflow exception (7.6.2.3).

— Floating environment macros other than `FE_DFL_ENV` that can be used as the argument to the `fesetenv` or `feupdateenv` function (7.6.4.3, 7.6.4.4).

— 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 or greater than 2 (7.12).

— The infinity to which the **INFINITY** macro expands, if any (7.12).

— The quiet NaN to which the **NAN** macro expands, when it is defined (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, and whether **errno** is set to the value of the macro **EDOM**, on domain errors (7.12.1).

— Whether the mathematics functions set **errno** to the value of the macro **ERANGE** on overflow and/or underflow range errors (7.12.1).

— The default state for the **FP_CONTRACT** pragma (7.12.2)

— Whether a domain error occurs or zero is returned when the **fmod** function has a second argument of zero (7.12.10.1).

— The base-2 logarithm of the modulus used by the **remquo** function in reducing the quotient (7.12.10.3).

— The set of signals, their semantics, and their default handling (7.14).

— If the equivalent of **signal(sig, SIG_DFL);** is not executed prior to the call of a signal handler, the blocking of the signal that is performed (7.14.1.1).

— Whether the equivalent of **signal(sig, SIG_DFL);** is executed prior to the call of a signal handler for the signal **SIGILL** (7.14.1.1).

— Signal values other than **SIGFPE**, **SIGILL**, and **SIGSEGV** that correspond to a computational exception (7.14.1.1).

— The null pointer constant to which the macro **NULL** expands (7.17).

— Whether the last line of a text stream requires a terminating new-line character (7.19.2).

— Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.19.2).

— The number of null characters that may be appended to data written to a binary stream (7.19.2).

— Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.19.3).

— Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3).

— The characteristics of file buffering (7.19.3).

— Whether a zero-length file actually exists (7.19.3).

— The rules for composing valid file names (7.19.3).

— Whether the same file can be open multiple times (7.19.3).

— The nature and choice of encodings used for multibyte characters in files (7.19.3).

— The effect of the **remove** function on an open file (7.19.4.1).

— The effect if a file with the new name exists prior to a call to the **rename** function (7.19.4.2).

— Whether an open temporary file is removed upon abnormal program termination (7.19.4.3).

— What happens when the **tmpnam** function is called more than **TMP_MAX** times (7.19.4.4).

— Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4).

— The style used to print an infinity or NaN, and the meaning of the *n-char-sequence* if that style is printed for a NaN (7.19.6.1, 7.24.2.1).

— The output for **%p** conversion in the **fprintf** or **fwprintf** function (7.19.6.1, 7.24.2.1).

— 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 (7.19.6.2, 7.24.2.1).

— The set of sequences matched by the **%p** conversion in the **fscanf** or **fwscanf** function (7.19.6.2, 7.24.2.2).

— The interpretation of the input item corresponding to a **%p** conversion in the **fscanf** or **fwscanf** function (7.19.6.2, 7.24.2.2).

— The value to which the macro **errno** is set by the **fgetpos**, **fsetpos**, or **ftell** functions on failure (7.19.9.1, 7.19.9.3, 7.19.9.4).

— The meaning of the *n-char-sequence* in a string converted by the **strtod**, **strtof**, **strtold**, **wcstod**, **wcstof**, or **wcstold** function (7.20.1.3, 7.24.4.1.1).

— Whether or not the **strtod**, **strtof**, **strtold**, **wcstod**, **wcstof**, or **wcstold** function sets **errno** to **ERANGE** when underflow occurs (7.20.1.3, 7.24.4.1.1).

— Whether the **calloc**, **malloc**, and **realloc** functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.20.3).

— Whether open output streams are flushed, open streams are closed, or temporary files are removed when the **abort** function is called (7.20.4.1).

— The termination status returned to the host environment by the **abort** function (7.20.4.1).

— The value returned by the **system** function when its argument is not a null pointer (7.20.4.5).

— The local time zone and Daylight Saving Time (7.23.1).

— The era for the **clock** function (7.23.2.1).

— The positive value for **tm_isdst** in a normalized **tmx** structure (7.23.2.6).

— The replacement string for the **%Z** specifier to the **strftime**, **strfxtime**, **wcsftime**, and **wcsfxtime** functions in the **"C"** locale (7.23.3.5, 7.23.3.6, (7.24.5.1, 7.24.5.2).

— Whether or when the trigonometric, hyperbolic, base-*e* exponential, base-*e* logarithmic, error, and log gamma functions raise the inexact exception in an IEC 60559 conformant implementation (F.9).

— Whether the inexact exception may be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (F.9).

— Whether the underflow (and inexact) exception may be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (F.9).

— Whether the functions honor the rounding direction mode (F.9).

K.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, 7.18.2, 7.18.3). *

— The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).

— The value of the result of the **sizeof** operator (6.5.3.4).

K.4 Locale-specific behavior

1 The following characteristics of a hosted environment are locale-specific and shall be documented by the implementation:

— Additional members of the execution character set beyond the required members (5.2.1).

— The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the required single-byte characters (5.2.1.2).

— The shift states used for the encoding of multibyte characters (5.2.1.2).

— The direction of writing of successive printable characters (5.2.2).

— The decimal-point character (7.1.1).

— The set of *printing characters* (7.4).

— The set of *control characters* (7.4).

— The sets of characters tested for by the **isalpha**, **islower**, **ispunct**, **isspace**, or **isupper** functions (7.4.1.2, 7.4.1.6, 7.4.1.8, 7.4.1.9, 7.4.1.10).

— The native environment (7.11.1.1).

— Additional subject sequences accepted by the string conversion functions (7.20.1) and the wide string numeric conversion function (7.24.4.1).

— The collation sequence of the execution character set (7.21.4.3).

— The contents of the error message strings set up by the **`strerror`** function (7.21.6.2).

— The formats for time and date (7.23.3.5).

— Character mappings that are supported by the **`towctrans`** function (7.25.1).

— Character classifications that are supported by the **`iswctype`** function (7.25.1).

— The set of *printing wide characters* (7.25.2).

— The set of *control wide characters* (7.25.2).

— The sets of wide characters tested for by the **`iswalpha`**, **`iswlower`**, **`iswpunct`**, **`iswspace`**, or **`iswupper`** functions (7.25.2.1.2, 7.25.2.1.6, 7.25.2.1.8, 7.25.2.1.9, 7.25.2.1.10).

K.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.

K.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).

K.5.2 Specialized identifiers

1 Characters other than the underscore `_`, letters, and digits, that are not defined in the required source character set (such as the dollar sign $, or characters in national character sets) may appear in an identifier (6.4.2).

K.5.3 Lengths and cases of identifiers

1 All characters in identifiers (with or without external linkage) are significant (6.4.2).

K.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).

K.5.5 Writable string literals

1 String literals are modifiable (in which case, identical string literals should denote distinct objects) (6.4.5).

K.5.6 Other arithmetic types

1 Additional arithmetic types, such as **`__int128`**, and their appropriate conversions are defined (6.2.5, 6.3.1.1).

K.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).

K.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).

K.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).

K.5.10 The **asm** keyword

1 The **asm** keyword may be used to insert assembly language directly into the translator output; the most common implementation is via a statement of the form

```
asm ( character-string-literal );
```

(6.8).

K.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).

K.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).

K.5.13 Extra arguments for signal handlers

1 Handlers for specific signals may be called with extra arguments in addition to the signal number (7.14.1.1).

K.5.14 Additional stream types and file-opening modes

1 Additional mappings from files to streams may be supported (7.19.2).

2 Additional file-opening modes may be specified by characters appended to the **mode** argument of the **fopen** function (7.19.5.3).

K.5.15 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 (7.19.7.11, 7.24.3.10).

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/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange*.
8. ISO/IEC 2382−1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*.
9. ISO 4217:1995, *Codes for the representation of currencies and funds*.
10. ISO 8601:1988, *Data elements and interchange formats — Information interchange — Representation of dates and times*.
11. ISO/IEC 9899:1990, *Programming languages — C*. *
12. ISO/IEC 9899/COR1:1994, *Technical Corrigendum 1*.
13. ISO/IEC 9899/COR2:1996, *Technical Corrigendum 2*.
14. ISO/IEC 9899/AMD1:1995, *Amendment 1 to ISO/IEC 9899:1990 C Integrity*.
15. ISO/IEC 9945−2:1993, *Information technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities*. |
16. ISO/IEC TR 10176:1998, *Information technology — Guidelines for the preparation of programming language standards*.
17. ISO/IEC 10646:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*. |
18. ISO/IEC 10967−1:1994, *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*.

Index