Contents

1 Scope 1

2 Compliance 1

3 Language 1
- 3.1 Operators 2
- 3.2 Punctuators 2
- 3.3 Version macro 2

4 Library 2
- 4.1 Definitions of terms 2
- 4.2 Standard headers 3
- 4.3 Errors <errno.h> 3
- 4.4 Alternative spellings <iso646.h> 3
- 4.5 Wide-character classification and mapping utilities <wctype.h> 3
  - 4.5.1 Introduction 3
  - 4.5.2 Wide-character classification utilities 4
    - 4.5.2.1 Wide-character classification functions 4
    - 4.5.2.2 Extensible wide-character classification functions 6
  - 4.5.3 Wide-character mapping utilities 7
    - 4.5.3.1 Wide-character case-mapping functions 7
    - 4.5.3.2 Extensible wide-character mapping functions 7
- 4.6 Extended multibyte and wide-character utilities <wchar.h> 8
  - 4.6.1 Introduction 8
  - 4.6.2 Input/output 9
    - 4.6.2.1 Streams 9
    - 4.6.2.2 Files 10
    - 4.6.2.3 Formatted input/output functions 10
    - 4.6.2.4 Formatted wide-character input/output functions 13
    - 4.6.2.5 Wide-character input/output functions 21
  - 4.6.3 General wide-string utilities 24
    - 4.6.3.1 Wide-string numeric conversion functions 24
    - 4.6.3.2 Wide-string copying functions 27
    - 4.6.3.3 Wide-string concatenation functions 27
    - 4.6.3.4 Wide-string comparison functions 28
    - 4.6.3.5 Wide-string search functions 29
    - 4.6.3.6 Wide-character array functions 31
  - 4.6.4 The wcsftime function 33
  - 4.6.5 Extended multibyte and wide-character conversion utilities 33
    - 4.6.5.1 Single-byte wide-character conversion functions 34
    - 4.6.5.2 The mbsinit function 34
    - 4.6.5.3 Restartable multibyte/wide-character conversion functions 34
    - 4.6.5.4 Restartable multibyte/wide-string conversion functions 36
- 4.7 Future library directions 37
  - 4.7.1 Wide-character classification and mapping utilities <wctype.h> 37
  - 4.7.2 Extended multibyte and wide-character utilities <wchar.h> 37

Annex A: Library summary (informative) 39

Annex B: Rationale (informative) 41

Index 53

Foreword

[to be supplied by ISO Secretariat]

Introduction

This document is the first amendment to the International Standard ISO/IEC 9899:1990, Programming Language — C. Although its purpose is to modify the base standard, this first amendment has been written, for the greater part, as a stand-alone document — one that need not be read side-by-side with the base standard.

This first amendment primarily consists of a set of library extensions that provide a complete and consistent set of utilities for application programming using multibyte and wide characters. It also contains extensions that provide alternate spellings for certain tokens.

The base standard deliberately chose not to include a complete multibyte and wide-character library. Instead, it defined just enough support to provide a firm foundation, both in the library and language proper, on which implementations and programming expertise could grow. Vendors did implement such extensions; this first amendment reflects the studied and careful inclusion of the best of today's existing art in this area.

The base standard also chose to provide only minimal support for writing C source code in character sets that redefine some of the punctuation characters, such as national variants of ISO 646. The alternate spellings provided here can be used to write many (but not all) tokens that are less readable when expressed in terms of trigraphs.

This first amendment to ISO/IEC 9899:1990 is divided into three major subdivisions:

— those additions and changes that affect the preliminary subdivision of ISO/IEC 9899:1990 (clauses 1 through 4);

— those additions and changes that affect the language syntax, constraints, and semantics (ISO/IEC 9899:1990 clause 6);

— those additions and changes that affect library facilities (ISO/IEC 9899:1990 clause 7).

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 first amendment. References are used to refer both to the base standard and to related subclauses within this document. These two can be distinguished either by context or are labeled as referring to the base standard (as above). Annex A summarizes the contents of this first amendment. Annex B provides a rationale. This introduction, the examples, the footnotes, the background, the references, the annexes, and the index do not form part of this first amendment.

Information Processing — Programming Languages C —

AMENDMENT 1

1 Scope

This amendment defines extensions to ISO/IEC 9899:1990 that provide a more complete set of multibyte and wide-character utilities, as well as alternative spellings for certain tokens. Use of these features can help promote international portability of C programs.

This amendment specifies extensions that affect various clauses of ISO/IEC 9899:1990:

— To the compliance clause (clause 4), the additional header **<iso646.h>** is provided by both freestanding and hosted implementations.

— To the language clause (clause 6), six additional tokens are accepted.

— To the library clause (clause 7), new capabilities are specified for the existing formatted input/output functions (7.9.6).

— To the library clause (clause 7), the additional header **<wctype.h>** is provided, which defines a macro, several types, and many functions, including:

- wide-character testing functions, **iswalnum** for example;
- extensible wide-character classification functions, **wctype** and **iswctype**;
- wide-character case-mapping functions, **towlower** and **towupper**;
- extensible wide-character case-mapping functions, **wctrans** and **towctrans**.

— To the library clause (clause 7), the additional header **<wchar.h>** is provided, which defines several macros, several types, and many functions, including:

- formatted wide-character input/output functions, **fwprintf** for example;
- wide-character input/output functions, **fgetwc** for example;
- wide-string numeric conversion functions, **wcstod** for example;
- wide-string general utility functions, **wcscpy** for example;
- a wide-string time conversion function, **wcsftime**;
- restartable multibyte/wide-character conversion functions, **mbrtowc** for example;
- restartable multibyte/wide-string conversion functions, **mbsrtowcs** and **wcsrtombs**.

2 Compliance

— Extensions to clause 4 —

The description is adjusted so that the standard header **<iso646.h>** is included in the list of headers that must be provided by both freestanding and hosted implementations.

Forward References: alternate spellings **<iso646.h>** (4.4).

3 Language

Subclauses 6.1.5 and 6.1.6 of ISO/IEC 9899:1990 are adjusted to include the following six additional tokens. In all aspects of the language, these six tokens

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

behave, respectively, the same as these existing six tokens

[ ] { } # ##

except for their spelling.[1)]

1) Thus **[** and **<:** behave differently when "stringized" (see ISO/IEC 9899:1990 subclause 6.8.3.2), but can otherwise be freely interchanged.

7

3.1 Operators

— Extensions to 6.1.5 —

Syntax

operator: also one of

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

Constraints

The operators `[ ]`, `( )`, and `? :` (independent of spelling) shall occur in pairs, possibly separated by expressions. The operators # and ## (also spelled `%:` and `%:%:`, respectively) shall occur in macro-defining preprocessing directives only.

3.2 Punctuators

— Extensions to 6.1.6 —

Syntax

punctuator: also one of

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

Constraints

The punctuators `[ ]`, `( )`, and `{ }` (independent of spelling) shall occur (after translation phase 4) in pairs, possibly separated by expressions, declarations, or statements. The punctuator # (also spelled `%:`) shall occur in preprocessing directives only.

3.3 Version macro

Subclause 6.8.8 is adjusted to include the following macro name defined by the implementation:

`__STDC_VERSION__`

which expands to the decimal constant `199409L`, intended to indicate an implementation conforming to this amendment.

4 Library

Various portions of clause 7 of ISO/IEC 9899:1990 are adjusted to include the following specifications. Each subclause to be extended usually has a correspondingly titled subclause in this portion of this amendment.

The identifiers with external linkage declared in either `<wctype.h>` or `<wchar.h>` which are not already reserved as identifiers with external linkage by ISO/IEC 9899:1990 are reserved for use as identifiers with external linkage only if at least one inclusion of either `<wctype.h>` or `<wchar.h>` occurs in one or more of the translation units that constitute the program.[2)]

4.1 Definitions of terms

— Extensions to 7.1.1 —

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.[3)]

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

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.

---

2) This behavior differs from those identifiers with external linkage associated with the headers listed in and referenced by ISO/IEC 9899:1990 subclauses 7.1.2 and 7.1.3, which are always reserved. Note that including either of these headers in a translation unit will affect other translation units in the same program, even though they do not include either header.

3) An equivalent definition can be found in subclause 6.1.3.4 of ISO/IEC 9899:1990.

A *shift sequence* is a contiguous sequence of bytes within a multibyte string that causes a change in shift state. (See ISO/IEC 9899:1990 subclause 5.2.1.2.) A shift sequences shall not have a corresponding wide character; it is instead taken to be an adjunct to an adjacent multibyte character.[4)]

4.2 Standard headers

— Extensions to 7.1.2 —

The list of standard headers is adjusted to include three new standard headers, **<iso646.h>**, **<wctype.h>**, and **<wchar.h>**.

4.3 Errors <errno.h>

— Extensions to 7.1.4 —

The list of macros defined in **<errno.h>** is adjusted to include a new macro, **EILSEQ**.

4.4 Alternative spellings <iso646.h>

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  ^=
```

4.5 Wide-character classification and mapping utilities <wctype.h>

4.5.1 Introduction

The header **<wctype.h>** declares three data types, one macro, and many functions.[5)]

The types declared are

wint_t

which is an integral type unchanged by integral 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).[6)]

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.

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.[7)] 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) For state-dependent encodings, the values for **MB_CUR_MAX** and **MB_LEN_MAX** must 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.

5) See "future library directions" (4.7.1).

6) **wchar_t** and **wint_t** can be the same integral type.

7) The value of the macro **WEOF** may differ from that of **EOF** and need not be negative.

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.

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.

The behavior of these functions is affected by the **LC_CTYPE** category of the current locale.

4.5.2 Wide-character classification utilities

The header **<wctype.h>** declares several functions useful for classifying wide characters.

The term *printing wide character* refers to a member of an implementation-defined 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 an implementation-defined set of wide characters that are not printing wide characters.

4.5.2.1 Wide-character classification functions

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.

Except for the **iswgraph** and **iswpunct** functions with respect to printing white-space wide characters other than **L' '**, each of the following eleven 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 respectively matching character testing function from ISO/IEC 9899:1990 subclause 7.3.1 returns true.[8)]

Forward References: the **wctob** function (4.6.5.1.2).

4.5.2.1.1 The iswalnum function

Synopsis

```
#include <wctype.h>
int iswalnum(wint_t wc);
```

Description

The **iswalnum** function tests for any wide character for which **iswalpha** or **iswdigit** is true.

4.5.2.1.2 The iswalpha function

Synopsis

```
#include <wctype.h>
int iswalpha(wint_t wc);
```

Description

The **iswalpha** function tests for any wide character for which **iswupper** or **iswlower** is true, or any wide character that is one of an implementation-defined set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

4.5.2.1.3 The iswcntrl function

Synopsis

```
#include <wctype.h>
int iswcntrl(wint_t wc);
```

Description

The **iswcntrl** function tests for any control wide character.

---

8) For example, if the expression **isalpha(wctob(wc))** evaluates to true, then the call **iswalpha(wc)** must also return 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)** must be true, but not both.

10

4.5.2.1.4 The `iswdigit` function

Synopsis

```
#include <wctype.h>
int iswdigit(wint_t wc);
```

Description

The `iswdigit` function tests for any wide character that corresponds to a decimal-digit character (as defined in ISO/IEC 9899:1990 subclause 5.2.1).

4.5.2.1.5 The `iswgraph` function

Synopsis

```
#include <wctype.h>
int iswgraph(wint_t wc);
```

Description

The `iswgraph` function tests for any wide character for which `iswprint` is true and `iswspace` is false.[9)]

4.5.2.1.6 The `iswlower` function

Synopsis

```
#include <wctype.h>
int iswlower(wint_t wc);
```

Description

The `iswlower` function tests for any wide character that corresponds to a lowercase letter or is one of an implementation-defined set of wide characters for which none of `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` is true.

4.5.2.1.7 The `iswprint` function

Synopsis

```
#include <wctype.h>
int iswprint(wint_t wc);
```

Description

The `iswprint` function tests for any printing wide character.

4.5.2.1.8 The `iswpunct` function

Synopsis

```
#include <wctype.h>
int iswpunct(wint_t wc);
```

Description

The `iswpunct` function tests for any printing wide character that is one of an implementation-defined set of wide characters for which neither `iswspace` nor `iswalnum` is true.

4.5.2.1.9 The `iswspace` function

Synopsis

```
#include <wctype.h>
int iswspace(wint_t wc);
```

Description

The `iswspace` function tests for any wide character that corresponds to an implementation-defined set of wide characters for which `iswalnum` is false.

---

9) Note that the behavior of the `iswgraph` and `iswpunct` functions may differ from their matching functions in ISO/IEC 9899:1990 subclause 7.3.1 with respect to printing white space basic execution characters other than ' '.

4.5.2.1.10 The `iswupper` function

Synopsis

```
#include <wctype.h>
int iswupper(wint_t wc);
```

Description

The **iswupper** function tests for any wide character that corresponds to an uppercase letter or is one of an implementation-defined set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

4.5.2.1.11 The `iswxdigit` function

Synopsis

```
#include <wctype.h>
int iswxdigit(wint_t wc);
```

Description

The **iswxdigit** function tests for any wide character that corresponds to a hexadecimal-digit character (as defined in ISO/IEC 9899:1990 subclause 6.1.3.2).

4.5.2.2 Extensible wide-character classification functions

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 (4.5.2.1).

4.5.2.2.1 The `wctype` function

Synopsis

```
#include <wctype.h>
wctype_t wctype(const char *property);
```

Description

The **wctype** function constructs a value with type **wctype_t** that describes a class of wide characters identified by the string argument, **property**.

The eleven strings listed in the description of the **iswctype** function shall be valid in all locales as **property** arguments to the **wctype** function.

Returns

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.

4.5.2.2.2 The `iswctype` function

Synopsis

```
#include <wctype.h>
int iswctype(wint_t wc, wctype_t desc);
```

Description

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**.

Each of the following eleven expressions has a truth-value equivalent to the call to the wide-character testing function (4.5.2.1) 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

The **iswctype** function returns nonzero (true) if and only if the value of the wide character **wc** has the property described by **desc**.

4.5.3 Wide-character mapping utilities

The header **<wctype.h>** declares several functions useful for mapping wide characters.

4.5.3.1 Wide-character case-mapping functions

4.5.3.1.1 The towlower function

Synopsis

```
#include <wctype.h>
wint_t towlower(wint_t wc);
```

Description

The **towlower** function converts an uppercase letter to the corresponding lowercase letter.

Returns

If the argument is a wide character for which **iswupper** is true and there is a corresponding wide character for which **iswlower** is true, the **towlower** function returns the corresponding wide character; otherwise, the argument is returned unchanged.

4.5.3.1.2 The towupper function

Synopsis

```
#include <wctype.h>
wint_t towupper(wint_t wc);
```

Description

The **towupper** function converts a lowercase letter to the corresponding uppercase letter.

Returns

If the argument is a wide character for which **iswlower** is true and there is a corresponding wide character for which **iswupper** is true, the **towupper** function returns the corresponding wide character; otherwise, the argument is returned unchanged.

4.5.3.2 Extensible wide-character mapping functions

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 (4.5.3.1).

4.5.3.2.1 The wctrans function

Synopsis

```
#include <wctype.h>
wctrans_t wctrans(const char *property);
```

Description

The **wctrans** function constructs a value with type **wctrans_t** that describes a mapping between wide characters identified by the string argument, **property**.

The two strings listed in the description of the **towctrans** function shall be valid in all locales as **property** arguments to the **wctrans** function.

Returns

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.

4.5.3.2.2 The `towctrans` function

Synopsis

```
#include <wctype.h>
wint_t towctrans(wint_t wc, wctrans_t desc);
```

Description

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 `towctrans` that returned the value `desc`.

Each of the following two expressions behaves the same as the call to the wide-character case-mapping function (4.5.3.1) in the comment that follows the expression:

```
towctrans(wc, wctrans("tolower"))     /* towlower(wc) */
towctrans(wc, wctrans("toupper"))     /* towupper(wc) */
```

Returns

The `towctrans` function returns the mapped value of `wc` using the mapping described by `desc`.

4.6 Extended multibyte and wide-character utilities <wchar.h>

4.6.1 Introduction

The header <wchar.h> declares four data types, one tag, four macros, and many functions.[10)]

The types declared are `wchar_t` and `size_t` (both described in ISO/IEC 9899:1990 subclause 7.1.6),

`mbstate_t`

which is a nonarray object type that can hold the conversion state information necessary to convert between sequences of multibyte characters and wide characters, and

`wint_t`

described in subclause 4.5.1.

The tag `tm` is declared as naming an incomplete structure type, the contents of which are described in ISO/IEC 9899:1990 subclause 7.12.1.

The macros defined are `NULL` (described in ISO/IEC 9899:1990 subclause 7.1.6),

`WCHAR_MAX`

which is the maximum value representable by an object of type `wchar_t`,[11)]

`WCHAR_MIN`

which is the minimum value representable by an object of type `wchar_t`, and

`WEOF`

described in subclause 4.5.2.

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;

— A function for wide-string date and time conversion; and

---

10) See "future library directions" (4.7.1).

11) The values `WCHAR_MAX` and `WCHAR_MIN` do not necessarily correspond to members of the extended character set.

— Functions that provide extended capabilities for conversion between multibyte and wide-character sequences.

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.

4.6.2 Input/output

— Extensions to 7.9.1 —

The header **<wchar.h>** declares a number of functions useful for wide-character input and output.

The wide-character input/output functions described in this subclause provide operations analogous to most of those described in ISO/IEC 9899:1990 subclause 7.9, 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 subclause 4.6.2.2, below.

The input/output functions described here and in ISO/IEC 9899:1990 are given the following collective terms:

— The *wide-character input functions* — those functions described in this subclause that perform input into wide characters and wide strings: **fgetwc**, **fgetws**, **getwc**, **getwchar**, **fwscanf**, and **wscanf**.

— The *wide-character output functions* — those functions described in this subclause 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* — the **ungetc** function and the input/output functions described in ISO/IEC 9899:1990 subclause 7.9: **fgetc**, **fgets**, **fprintf**, **fputc**, **fputs**, **fread**, **fscanf**, **fwrite**, **getc**, **getchar**, **gets**, **printf**, **putc**, **putchar**, **puts**, **scanf**, **vfprintf**, and **vprintf**.

4.6.2.1 Streams

— Extensions to 7.9.2 —

The definition of a stream is adjusted to include an *orientation* for both text and binary streams. 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 *wide-oriented*. Similarly, once a byte input/output function has been applied to a stream without orientation, the stream becomes *byte-oriented*. 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.)

Byte input/output functions shall not be applied to a wide-oriented stream; and wide-character input/output functions shall not be applied to a byte-oriented stream. The remaining stream operations do not affect and are not affected by a stream's orientation, except for the following additional restrictions:

— Binary wide-oriented streams have the file-positioning restrictions ascribed to both text and binary streams.

— For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide-character output function can overwrite a partial multibyte character; any file contents beyond the byte(s) written are henceforth undefined.

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.

Returns

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.

4.5.3.2.2 The `towctrans` function

Synopsis

```
#include <wctype.h>
wint_t towctrans(wint_t wc, wctrans_t desc);
```

Description

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 `towctrans` that returned the value `desc`.

Each of the following two expressions behaves the same as the call to the wide-character case-mapping function (4.5.3.1) in the comment that follows the expression:

```
towctrans(wc, wctrans("tolower"))     /* towlower(wc) */
towctrans(wc, wctrans("toupper"))     /* towupper(wc) */
```

Returns

The `towctrans` function returns the mapped value of `wc` using the mapping described by `desc`.

4.6 Extended multibyte and wide-character utilities <wchar.h>

4.6.1 Introduction

The header <wchar.h> declares four data types, one tag, four macros, and many functions.[10)]

The types declared are `wchar_t` and `size_t` (both described in ISO/IEC 9899:1990 subclause 7.1.6),

`mbstate_t`

which is a nonarray object type that can hold the conversion state information necessary to convert between sequences of multibyte characters and wide characters, and

`wint_t`

described in subclause 4.5.1.

The tag `tm` is declared as naming an incomplete structure type, the contents of which are described in ISO/IEC 9899:1990 subclause 7.12.1.

The macros defined are `NULL` (described in ISO/IEC 9899:1990 subclause 7.1.6),

`WCHAR_MAX`

which is the maximum value representable by an object of type `wchar_t`,[11)]

`WCHAR_MIN`

which is the minimum value representable by an object of type `wchar_t`, and

`WEOF`

described in subclause 4.5.2.

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;

— A function for wide-string date and time conversion; and

---

10) See "future library directions" (4.7.1).

11) The values `WCHAR_MAX` and `WCHAR_MIN` do not necessarily correspond to members of the extended character set.

If an `l` qualifier is present, the argument shall be a pointer to an array of `wchar_t` type. Wide characters from the array are converted to multibyte characters (each as if by a call to the `wcrtomb` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many 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.[13)]

The above extension is applicable to all the formatted output functions specified in ISO/IEC 9899:1990.

Examples

The examples are adjusted to include the following:

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.

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, "|%13.1ls|\n", wstr);
fprintf(stdout, "|%13.15ls|\n", &wstr[2]);
fprintf(stdout, "|%13lc|\n", wstr[5]);
```

will print the following seven lines:

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

Forward References: conversion state (4.6.5), the `wcrtomb` function (4.6.5.3.3).

4.6.2.3.2 The `fscanf` function

— Extensions to 7.9.6.2 —

Adjust the description of the qualifiers `h`, `l`, and `L` to include the additional sentences:

The conversion specifiers `c`, `s`, and `[` shall be preceded by `l` if the corresponding argument is a pointer to `wchar_t` rather than a pointer to a character type.

Replace the definition of directive failure (page 135, lines 34-36, beginning with, "If the length of the input item is zero...") with:

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.

Replace the description of the `s` conversion specifier with:

`s` Matches a sequence of non-white-space characters.[14)] If no `l` qualifier is present, the corresponding argument shall be a pointer to a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

---

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

If an `l` qualifier 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 an array of `wchar_t` large enough to accept the sequence and the terminating null wide character, which will be added automatically.

Replace the first two sentences of the description of the `[` conversion specifier with:

`[` Matches a nonempty sequence of characters from a set of expected characters (the *scanset*). If no `l` qualifier is present, the corresponding argument shall be a pointer to a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an `l` qualifier 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 an array of `wchar_t` large enough to accept the sequence and the terminating null wide character, which will be added automatically.

Replace the description of the `c` conversion specifier with:

`c` Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). If no `l` qualifier is present, the corresponding argument shall be a pointer to a character array large enough to accept the sequence. No null character is added.

If an `l` qualifier 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.

The above extension is applicable to all the formatted input functions specified in ISO/IEC 9899:1990.

Examples

The examples are adjusted to include the following:

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.

1. 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.

2. In contrast, after the call:

```
#include <stdio.h>
#include <stddef.h>
/*...*/
```

14) No special provisions are made for multibyte characters in the matching rules used by any of the conversion specifiers `s`, `[`, or `c` — the extent of the input field is still determined on a byte-by-byte basis. The resulting field must nevertheless be a sequence of multibyte characters that begins in the initial shift state.

— 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 **e**, **E**, 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** conversion. 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 **l** (ell) specifying that a following **c** conversion specifier applies to a **wint_t** argument; an optional **l** specifying that a following **s** conversion specifier applies to a pointer to a **wchar_t** argument; an optional **h** specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument is promoted according to the integral promotions, and its value is converted to **short int** or **unsigned short int** before printing); an optional **h** specifying that a following **n** conversion specifier applies to a pointer to a **short int** argument; an optional **l** specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long int** or **unsigned long int** argument; an optional **l** specifying that a following **n** conversion specifier applies to a pointer to a **long int** argument; or an optional **L** specifying that a following **e**, **E**, **f**, **g**, or **G** conversion specifier applies to a **long double** argument. If an **h**, **l**, or **L** appears with any other conversion specifier, the behavior is undefined.

— A wide character that specifies the type of conversion to be applied.

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.

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.)

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 to be converted to an "alternate form." For **o** conversion, it increases the precision to force the first digit of the result to be a zero, if necessary. For **x** (or **X**) conversion, a nonzero result has **0x** (or **0X**) prefixed to it. For **e**, **E**, **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* be removed from the result. For other conversions, the behavior is undefined.

0 For **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **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.

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.

```
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.

3. 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.

4. 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: conversion state (4.6.5), the `wcrtomb` function (4.6.5.3.3).

4.6.2.4 Formatted wide-character input/output functions

4.6.2.4.1 The `fwprintf` function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
int fwprintf(FILE *stream, const wchar_t *format, ...);
```

Description

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.

The format is composed of zero or more directives: ordinary wide characters (not `%`), and conversion specifications. The processing of conversion specifications is as if they were replaced in the format string by wide-character strings that are each the result of fetching zero or more subsequent arguments and converting them, if applicable, according to the corresponding conversion specifier. The expanded wide-character format string is then written to the output stream.

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.[15)]

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

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 The **double** argument 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.

e,E The **double** argument is converted in the style *[–]d.dde±dd*, where there is one digit before the decimal-point wide character (which is nonzero if the argument is nonzero) 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. If the value is zero, the exponent is zero.

g,G The **double** argument is converted in style **f** or **e** (or in style **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; a decimal-point wide character appears only if it is followed by a digit.

c If no **l** qualifier is present, the **int** argument is converted to a wide character as if by calling **btowc** and the resulting wide character is written. Otherwise, the **wint_t** argument is converted to **wchar_t** and written.

s If no **l** qualifier is present, the argument shall be a pointer to a character array containing a multibyte 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** qualifier is present, the argument shall be a pointer to 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 an 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.

% A **%** wide character is written. No argument is converted. The complete conversion specification shall be **%%**.

If a conversion specification is invalid, the behavior is undefined.[16)]

If any argument is, or points to, a union or an aggregate (except for an array of **char** type using **%s** conversion, an array of **wchar_t** type using **%ls** conversion, or a pointer using **%p** conversion), the behavior is undefined.

16) See "future library directions" (4.7.1).

21

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.

Returns

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

Environmental limit

The minimum value for the maximum number of wide characters produced by any single conversion shall be 509.

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 (4.6.5.1.1), the **mbrtowc** function (4.6.5.3.2).

4.6.2.4.2 The fwscanf function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
int fwscanf(FILE *stream, const wchar_t *format, ...);
```

Description

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.

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 a **%**. 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 **h**, **l** (ell), or **L** indicating the size of the receiving object. The conversion specifiers **c**, **s**, and **[** shall be preceded by **l** if the corresponding argument is a pointer to **wchar_t** rather than a pointer to a character type. The conversion specifiers **d**, **i**, and **n** shall be preceded by **h** if the corresponding argument is a pointer to **short int** rather than a pointer to **int**, or by **l** if it is a pointer to **long int**. Similarly, the conversion specifiers **o**, **u**, and **x** shall be preceded by **h** if the corresponding argument is a pointer to **unsigned short int** rather than a pointer to **unsigned int**, or by **l** if it is a pointer to **unsigned long int**. Finally, the conversion specifiers **e**, **f**, and **g** shall be preceded by **l** if the corresponding argument is a pointer to **double** rather than a pointer to **float**, or by **L** if it is a pointer to **long double**. If an **h**, **l**, or **L** appears with any other conversion specifier, the behavior is undefined.

— A wide character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The **fwscanf** function executes each directive of the format in turn. If a directive fails, as detailed below, the **fwscanf** function returns. Failures are described as input failures (if an encoding error occurs or due to the unavailability of input characters), or matching failures (due to inappropriate input).

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.

A directive that is an ordinary wide character is executed by reading the next wide character of the stream. If the wide character differs from the directive, the directive fails, and the differing and subsequent wide characters remain unread.

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:

Input white-space wide characters (as specified by the **iswspace** function) are skipped, unless the specification includes a **c** or **n** specifier.[17)]

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, not exceeding any specified field width, which is, or is a prefix of, a matching 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.

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 space provided, the behavior is undefined.

The following conversion specifiers are valid:

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 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 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.

e,f,g Matches an optionally signed floating-point number, whose format is the same as expected for the subject sequence of the **wcstod** function. The corresponding argument shall be a pointer to floating.

s Matches a sequence of non-white-space wide characters. If no **l** qualifier 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 a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

---

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

Otherwise, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** type large enough to accept the sequence and a 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** qualifier 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 a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an **l** qualifier is present, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** type large enough to accept the sequence and a 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 wide character (**]**). The wide characters between the brackets (the *scanlist*) comprise 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 right bracket wide character is the matching right bracket that ends the specification; otherwise the first 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.

c Matches a sequence of wide characters of the number specified by the field width (1 if no field width is present in the directive). If no **l** qualifier 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 a character array large enough to accept the sequence. No null character is added.

If an **l** qualifier is present, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** type large enough to accept the sequence. No null wide character is added.

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 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 affect the assignment count returned at the completion of execution of the **fwscanf** function.

% Matches a single **%**; no conversion or assignment occurs. The complete conversion specification shall be **%%**

If a conversion specification is invalid, the behavior is undefined.[18)]

The conversion specifiers **E**, **G**, and **X** are also valid and behave the same as, respectively, **e**, **g**, and **x**.

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.

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.

---

18) See "future library directions" (4.7.1).

Returns

The **fwscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **fwscanf** 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.

Examples

1. The call:

```
#include <stdio.h>
#include <wchar.h>
/*...*/
int n, i; float x; wchar_t name[50];
n = fwscanf(stdin, L"%d%f%ls", &i, &x, name);
```

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**.

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** function (4.6.3.1.1), the **wcstol** function (4.6.3.1.2), the **wcstoul** function (4.6.3.1.3), the **wcrtomb** function (4.6.5.3.3).

4.6.2.4.3 The wprintf function

Synopsis

```
#include <wchar.h>
int wprintf(const wchar_t *format, ...);
```

Description

The **wprintf** function is equivalent to **fwprintf** with the argument **stdout** interposed before the arguments to **wprintf**.

Returns

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

4.6.2.4.4 The wscanf function

Synopsis

```
#include <wchar.h>
int wscanf(const wchar_t *format, ...);
```

Description

The **wscanf** function is equivalent to **fwscanf** with the argument **stdin** interposed before the arguments to **wscanf**.

Returns

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.

4.6.2.4.5 The swprintf function

Synopsis

```
#include <wchar.h>
int swprintf(wchar_t *s, size_t n, const wchar_t *format, ...);
```

Description

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

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 **n** or more wide characters were requested to be written.

4.6.2.4.6 The swscanf function

Synopsis

```
#include <wchar.h>
int swscanf(const wchar_t *s, const wchar_t *format, ...);
```

Description

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

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.

4.6.2.4.7 The vfwprintf function

Synopsis

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);
```

Description

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.[19)]

Returns

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

Example

The following shows the use of the **vfwprintf** function in a general error-reporting routine.

```
#include <stdarg.h>
#include <stdio.h>
```

19) As the functions **vfwprintf**, **vswprintf**, and **vwprintf** invoke the **va_arg** macro, the value of **arg** after the return is unspecified.

```
#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);
}
```

4.6.2.4.8 The vwprintf function

Synopsis

```
#include <stdarg.h>
#include <wchar.h>
int vwprintf(const wchar_t *format, va_list arg);
```

Description

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.

Returns

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

4.6.2.4.9 The vswprintf function

Synopsis

```
#include <stdarg.h>
#include <wchar.h>
int vswprintf(wchar_t *s, size_t n, const wchar_t *format,
    va_list arg);
```

Description

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.

Returns

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 **n** or more wide characters were requested to be generated.

4.6.2.5 Wide-character input/output functions

4.6.2.5.1 The fgetwc function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *stream);
```

Description

The **fgetwc** function obtains the next wide character (if present) from the input stream pointed to by **stream**, and advances the associated file position indicator for the stream (if defined).

Returns

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**.[20)]

4.6.2.5.2 The fgetws function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t *s, int n, FILE *stream);
```

description

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

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.

4.6.2.5.3 The fputwc function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t fputwc(wchar_t c, FILE *stream);
```

Description

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

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**.

4.6.2.5.4 The fputws function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
int fputws(const wchar_t *s, FILE *stream);
```

Description

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

The **fputws** function returns **EOF** if a write or encoding error occurs; otherwise, it returns a nonnegative value.

---

20) 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.

4.6.2.5.5 The getwc function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *stream);
```

Description

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

The **getwc** function returns the next wide character from the input stream pointed to by **stream** or **WEOF**.

4.6.2.5.6 The getwchar function

Synopsis

```
#include <wchar.h>
wint_t getwchar(void);
```

Description

The **getwchar** function is equivalent to **getwc** with the argument **stdin**.

Returns

The **getwchar** function returns the next wide character from the input stream pointed to by **stdin** or **WEOF**.

4.6.2.5.7 The putwc function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wchar_t c, FILE *stream);
```

Description

The **putwc** function is equivalent to **fputwc**, 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

The **putwc** function returns the wide character written or **WEOF**.

4.6.2.5.8 The putwchar function

Synopsis

```
#include <wchar.h>
wint_t putwchar(wchar_t c);
```

Description

The **putwchar** function is equivalent to **putwc** with the second argument **stdout**.

Returns

The **putwchar** function returns the character written or **WEOF**.

4.6.2.5.9 The ungetwc function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t ungetwc(wint_t c, FILE *stream);
```

Description

The **ungetwc** function pushes the wide character specified by **c** back onto the input stream pointed to by **stream**. The 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.

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** 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.

If the value of **c** equals that of the macro **WEOF**, the operation fails and the input stream is unchanged.

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

The **ungetwc** function returns the wide character pushed back, or **WEOF** if the operation fails.

4.6.2.5.10 The fwide function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
int fwide(FILE *stream, int mode);
```

Description

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.[21] Otherwise, **mode** is zero and the function does not alter the orientation of the stream.

Returns

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.

4.6.3 General wide-string utilities

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.

4.6.3.1 Wide-string numeric conversion functions

4.6.3.1.1 The wcstod function

Synopsis

```
#include <wchar.h>
double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

21) If the orientation of the stream has already been determined, **fwide** does not change it.

Description

The **wcstod** function converts the initial portion of the wide string pointed to by **nptr** to **double** representation. First, it decomposes 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; and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, it attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then a nonempty sequence of digits optionally containing a decimal-point wide character, then an optional exponent part as defined for the corresponding single-byte characters in ISO/IEC 9899:1990 subclause 6.1.3.1, but no floating suffix. 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, a digit, or a decimal-point wide character.

If the subject sequence has the expected form, 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 ISO/IEC 9899:1990 subclause 6.1.3.1, except that the decimal-point wide character is used in place of a period, and that if neither an exponent part nor a decimal-point wide character appears, a decimal point is assumed to follow the last digit in the wide string. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Additional implementation-defined subject sequences may be accepted.

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

The **wcstod** 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, plus or minus **HUGE_VAL** is returned (according to the sign of the value), and the value of the macro **ERANGE** is stored in **errno**. If the correct value would cause underflow, zero is returned and the value of the macro **ERANGE** is stored in **errno**.

4.6.3.1.2 The wcstol function

Synopsis

```
#include <wchar.h>
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
```

Description

The **wcstol** function converts the initial portion of the wide string pointed to by **nptr** to **long int** representation. First, it decomposes 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, it attempts to convert the subject sequence to an integer, and returns the result.

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 ISO/IEC 9899:1990 subclause 6.1.3.2, 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.

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.

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 ISO/IEC 9899:1990 subclause 6.1.3.2. 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. A pointer to the final wide string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Additional implementation-defined subject sequences may be accepted.

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

The **wcstol** 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, **LONG_MAX** or **LONG_MIN** is returned (according to the sign of the value), and the value of the macro **ERANGE** is stored in **errno**.

4.6.3.1.3 The wcstoul function

Synopsis

```
#include <wchar.h>
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr,
    int base);
```

Description

The **wcstoul** function converts the initial portion of the wide string pointed to by **nptr** to **unsigned long int** representation. First, it decomposes 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 unsigned 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, it attempts to convert the subject sequence to an unsigned integer, and returns the result.

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 ISO/IEC 9899:1990 subclause 6.1.3.2, 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.

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.

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 ISO/IEC 9899:1990 subclause 6.1.3.2. 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. A pointer to the final wide string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Additional implementation-defined subject sequences may be accepted.

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

The **wcstoul** 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, **ULONG_MAX** is returned, and the value of the macro **ERANGE** is stored in **errno**.

4.6.3.2 Wide-string copying functions

4.6.3.2.1 The wcscpy function

Synopsis

```
#include <wchar.h>
wchar_t *wcscpy(wchar_t *s1, const wchar_t *s2);
```

Description

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

The **wcscpy** function returns the value of **s1**.

4.6.3.2.2 The wcsncpy function

Synopsis

```
#include <wchar.h>
wchar_t *wcsncpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

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**.[22]

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

The **wcsncpy** function returns the value of **s1**.

4.6.3.3 Wide-string concatenation functions

4.6.3.3.1 The wcscat function

Synopsis

```
#include <wchar.h>
wchar_t *wcscat(wchar_t *s1, const wchar_t *s2);
```

Description

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

The **wcscat** function returns the value of **s1**.

4.6.3.3.2 The wcsncat function

Synopsis

```
#include <wchar.h>
wchar_t *wcsncat(wchar_t *s1, const wchar_t *s2, size_t n);
```

---

22) 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.

Description

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.[23)]

Returns

The **wcsncat** function returns the value of **s1**.

4.6.3.4 Wide-string comparison functions

Unless explicitly stated otherwise, the functions described in this subclause order two wide characters the same way as two integers of the underlying integral type designated by **wchar_t**.

4.6.3.4.1 The wcscmp function

Synopsis

```
#include <wchar.h>
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

Description

The **wcscmp** function compares the wide string pointed to by **s1** to the wide string pointed to by **s2**.

Returns

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**.

4.6.3.4.2 The wcscoll function

Synopsis

```
#include <wchar.h>
int wcscoll(const wchar_t *s1, const wchar_t *s2);
```

Description

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

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.

4.6.3.4.3 The wcsncmp function

Synopsis

```
#include <wchar.h>
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

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

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**.

---

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

4.6.3.4.4 The `wcsxfrm` function

Synopsis

```
#include <wchar.h>
size_t wcsxfrm(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

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

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.

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)
```

4.6.3.5 Wide-string search functions

4.6.3.5.1 The `wcschr` function

Synopsis

```
#include <wchar.h>
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

Description

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

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.

4.6.3.5.2 The `wcscspn` function

Synopsis

```
#include <wchar.h>
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

Description

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

The **wcscspn** function returns the length of the segment.

4.6.3.5.3 The `wcspbrk` function

Synopsis

```
#include <wchar.h>
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
```

Description

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

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**.

4.6.3.5.4 The wcsrchr function

Synopsis

```
#include <wchar.h>
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
```

Description

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

The **wcsrchr** function returns a pointer to the wide character, or a null pointer if **c** does not occur in the wide string.

4.6.3.5.5 The wcsspn function

Synopsis

```
#include <wchar.h>
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

Description

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

The **wcsspn** function returns the length of the segment.

4.6.3.5.6 The wcsstr function

Synopsis

```
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
```

Description

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

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**.

4.6.3.5.7 The wcstok function

Synopsis

```
#include <wchar.h>
wchar_t *wcstok(wchar_t *s1, const wchar_t *s2, wchar_t **ptr);
```

Description

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.

For the first call in the sequence, **s1** shall point to a wide string, while in subsequent calls for the same string, **s1** shall be a null pointer. If **s1** is a null pointer, the value pointed to by **ptr** shall match that set by the previous call for the same wide string; otherwise its value is ignored. The separator wide string pointed to by **s2** may be different from call to call.

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.

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.

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 end of the previously returned token (if any).

Returns

The **wcstok** function returns a pointer to the first wide character of a token, or a null pointer if there is no token.

Example

```
#include <wchar.h>
static wchar_t str1[] = L"?a???b,,,#c";
static wchar_t str2[] = L"\t \t";
wchar_t *t, *ptr1, *ptr2;

t = wcstok(str1, L"?", &ptr1);   /* t points to the token L"a" */
t = wcstok(NULL, L",", &ptr1);   /* t points to the token L"??b" */
t = wcstok(str2, L" \t", &ptr2); /* t is a null pointer */
t = wcstok(NULL, L"#,", &ptr1);  /* t points to the token L"c" */
t = wcstok(NULL, L"?", &ptr1);   /* t is a null pointer */
```

4.6.3.5.8 The wcslen function

Synopsis

```
#include <wchar.h>
size_t wcslen(const wchar_t *s);
```

Description

The **wcslen** function computes the length of the wide string pointed to by **s**.

Returns

The **wcslen** function returns the number of wide characters that precede the terminating null wide character.

4.6.3.6 Wide-character array functions

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.

Unless explicitly stated otherwise, the functions described in this subclause order two wide characters the same way as two integers of the underlying integral type designated by **wchar_t**.

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 must still have valid values, as described in subclause 7.1.7 of ISO/IEC 9899:1990. On such a call, a function that copies wide characters copies zero wide characters, while a function that compares two wide character sequences returns zero.

4.6.3.6.1 The wmemchr function

Synopsis

```
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

Description

The **wmemchr** function locates the first occurrence of **c** in the initial **n** wide characters of the object pointed to by **s**.

Returns

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.

4.6.3.6.2 The wmemcmp function

Synopsis

```
#include <wchar.h>
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

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

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**.

4.6.3.6.3 The wmemcpy function

Synopsis

```
#include <wchar.h>
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

The **wmemcpy** function copies **n** wide characters from the object pointed to by **s2** to the object pointed to by **s1**.

Returns

The **wmemcpy** function returns the value of **s1**.

4.6.3.6.4 The wmemmove function

Synopsis

```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

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

The **wmemmove** function returns the value of **s1**.

4.6.3.6.5 The wmemset function

Synopsis

```
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

Description

The **wmemset** function copies the value of **c** into each of the first **n** wide characters of the object pointed to by **s**.

Returns

The **wmemset** function returns the value of **s**.

4.6.4 The wcsftime function

Synopsis

```
#include <wchar.h>
size_t wcsftime(wchar_t *s, size_t maxsize,
    const wchar_t *format, const struct tm *timeptr);
```

Description

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

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.

4.6.5 Extended multibyte and wide-character conversion utilities

The header **<wchar.h>** declares an extended set of functions useful for conversion between multibyte characters and wide characters.

Most of the following functions — those that are listed as "restartable," subclauses 4.6.5.3 and 4.6.5.4 — 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.

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.[24)]

On entry, each function takes the described conversion state (either internal or pointed to by **ps**) 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.

---

24) 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.

4.6.5.1 Single-byte wide-character conversion functions

4.6.5.1.1 The `btowc` function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t btowc(int c);
```

Description

The `btowc` function determines whether `c` constitutes a valid (one-byte) multibyte character in the initial shift state.

Returns

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.

4.6.5.1.2 The `wctob` function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
int wctob(wint_t c);
```

Description

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

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.

4.6.5.2 The `mbsinit` function

Synopsis

```
#include <wchar.h>
int mbsinit(const mbstate_t *ps);
```

Description

If `ps` is not a null pointer, the `mbsinit` function determines whether the pointed-to `mbstate_t` object describes an initial conversion state.

Returns

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.

4.6.5.3 Restartable multibyte/wide-character conversion functions

These functions differ from the corresponding multibyte character functions of ISO/IEC 9899:1990 subclause 7.10.7 (`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`.

Also unlike their corresponding functions, the return value does not represent whether the encoding is state-dependent.

4.6.5.3.1 The mbrlen function

Synopsis

```
#include <wchar.h>
size_t mbrlen(const char *s, size_t n, mbstate_t *ps);
```

Description

The **mbrlen** function is equivalent to the call:

```
mbrtowc((wchar_t *)0, s, n, ps != NULL ? ps : &internal)
```

where ***internal*** is the **mbstate_t** object for the **mbrlen** function.

Returns

The **mbrlen** function returns **(size_t)-2**, **(size_t)-1**, a value between zero and **n**, inclusive.

Forward References: the **mbrtowc** functions (4.5.6.3.2).

4.6.5.3.2 The mbrtowc function

Synopsis

```
#include <wchar.h>
size_t mbrtowc(wchar_t *pwc, const char *s, size_t n,
    mbstate_t *ps);
```

Description

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.

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

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).[25]

(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.

4.6.5.3.3 The wcrtomb function

Synopsis

```
#include <wchar.h>
size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps);
```

Description

If **s** is a null pointer, the **wcrtomb** function is equivalent to the call

---

25) 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).

41

```
wcrtomb(buf, L'\0', ps)
```

where *buf* is an internal buffer.

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 bytes is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

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.

4.6.5.4 Restartable multibyte/wide-string conversion functions

These functions differ from the corresponding multibyte string functions of ISO/IEC 9899:1990 subclause 7.10.8 (**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**.

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.

4.6.5.4.1 The mbsrtowcs function

Synopsis

```
#include <wchar.h>
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len,
    mbstate_t *ps);
```

Description

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**.[26] Each conversion takes place as if by a call to the **mbrtowc** function.

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

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. Otherwise, it returns the number of multibyte characters successfully converted, not including the terminating null (if any).

---

26) Thus, the value of **len** is ignored if **dst** is a null pointer.

4.6.5.4.2 The `wcsrtombs` function

Synopsis

```
#include <wchar.h>
size_t wcsrtombs(char *dst, const wchar_t **src, size_t len,
    mbstate_t *ps);
```

Description

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 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.[27)]

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

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 characters sequence, not including the terminating null (if any).

4.7 Future library directions

— Extension to 7.13 —

The list of headers and their reserved identifiers is adjusted to include the following:

4.7.1 Wide-character classification and mapping utilities <wctype.h>

Function names that begin with **is** or **to** and a lowercase letter (followed by any combination of digits, letters, and underscore) may be added to the declarations in the **<wctype.h>** header.

4.7.2 Extended multibyte and wide-character utilities <wchar.h>

Function names that begin with **wcs** and a lowercase letter (followed by any combination of digits, letters, and underscore) may be added to the declarations in the **<wchar.h>** header.

Lowercase letters may be added to the conversion specifiers in **fwprintf** and **fwscanf**.

---

27) If conversion stops because a terminating null character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte.

```
wcrtomb(buf, L'\0', ps)
```

where *buf* is an internal buffer.

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 bytes is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

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.

4.6.5.4 Restartable multibyte/wide-string conversion functions

These functions differ from the corresponding multibyte string functions of ISO/IEC 9899:1990 subclause 7.10.8 (**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**.

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.

4.6.5.4.1 The mbsrtowcs function

Synopsis

```
#include <wchar.h>
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len,
    mbstate_t *ps);
```

Description

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**.[26] Each conversion takes place as if by a call to the **mbrtowc** function.

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

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. Otherwise, it returns the number of multibyte characters successfully converted, not including the terminating null (if any).

---

26) Thus, the value of **len** is ignored if **dst** is a null pointer.

Annex A: Library summary (informative)

A.1 Errors <errno.h>

```
EILSEQ
```

A.2 Alternative spellings <iso646.h>

```
and
and_eq
bitand
bitor
compl
not
not_eq
or
or_eq
xor
xor_eq
```

A.3 Wide-character classification and mapping utilities <wctype.h>

```
wctrans_t
wctype_t
WEOF
wint_t
```

A.3.1 Wide-character classification functions

```
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);
wctype_t wctype(const char *property);
int iswctype(wint_t wc, wctype_t desc);
```

A.3.2 Wide-character mapping functions

```
wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
wctrans_t wctrans(const char *property);
wint_t towctrans(wint_t wc, wctrans_t desc);
```

A.4 Extended multibyte and wide-character utilities <wchar.h>

```
mbstate_t
size_t
struct tm
wchar_t
WCHAR_MAX
WCHAR_MIN
WEOF
wint_t
```

A.4.1 Formatted wide-character input/output functions

```
int fwprintf(FILE *stream, const wchar_t *format, ...);
int fwscanf(FILE *stream, const wchar_t *format, ...);
int wprintf(const wchar_t *format, ...);
int wscanf(const wchar_t *format, ...);
int swprintf(wchar_t *s, size_t n, const wchar_t *format, ...);
int swscanf(const wchar_t *s, const wchar_t *format, ...);
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);
```

```
int vwprintf(const wchar_t *format, va_list arg);
int vswprintf(wchar_t *s, size_t n, const wchar_t *format,
    va_list arg);
```

A.4.2 Wide-character input/output functions

```
wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t *s, int n, FILE *stream);
wint_t fputwc(wchar_t c, FILE *stream);
int fputws(const wchar_t *s, FILE *stream);
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);
int fwide(FILE *stream, int mode);
```

A.4.3 Wide-string numeric conversion functions

```
double wcstod(const wchar_t *nptr, wchar_t **endptr);
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr,
    int base);
```

A.4.4 Wide-string functions

```
wchar_t *wcscpy(wchar_t *s1, const wchar_t *s2);
wchar_t *wcsncpy(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wcscat(wchar_t *s1, const wchar_t *s2);
wchar_t *wcsncat(wchar_t *s1, const wchar_t *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 *s1, const wchar_t *s2, size_t n);
wchar_t *wcschr(const wchar_t *s, wchar_t c);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcstok(wchar_t *s1, const wchar_t *s2, wchar_t **ptr);
size_t wcslen(const wchar_t *s);
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *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);
```

A.4.5 Wide-string time conversion function

```
size_t wcsftime(wchar_t *s, size_t maxsize,
    const wchar_t *format, const struct tm *timeptr);
```

A.4.6 Extended multibyte/wide-character conversion functions

```
wint_t btowc(int c);
int wctob(wint_t c);
int mbsinit(const mbstate_t *ps);
size_t mbrlen(const char *s, size_t n, mbstate_t *ps);
size_t mbrtowc(wchar_t *pwc, const char *s, size_t n,
    mbstate_t *ps);
size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps);
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len,
    mbstate_t *ps);
size_t wcsrtombs(char *dst, const wchar_t **src, size_t len,
    mbstate_t *ps);
```

Annex B: Rationale (informative)

B.1 Background

Most traditional computer systems and computer languages, including traditional C, have an assumption (sometimes undocumented) that a "character" can be handled as an atomic quantity associated with a single memory storage unit — a "byte" or something similar. This is not true in general. For example, a Japanese, Chinese, or Korean character usually requires a representation of two or three bytes; this is a *multibyte character* as defined by ISO/IEC 9899:1990 subclause 3.13. Even in the Latin world, a multibyte coded character set may appear in the near future. This conflict is called a *byte and character problem.*

A related concern in this area is how to address having at least two different meanings for string length: number of bytes and number of characters.

To cope with these problems, many technical experts, particularly in Japan, have developed their own sets of additional multibyte character functions, sometimes independently and sometimes cooperatively. Fortunately, the developed extensions are actually quite similar. It can be said that in the process they have found common features for multibyte character support. Moreover, the industry currently has many good implementations of such support.

The above in no way denigrates the important groundwork in multibyte and wide-character programming provided by ISO/IEC 9899:1990.

— Both the source and execution character sets can contain multibyte characters (with possibly different encodings), even in the "C" locale.

— Multibyte characters are permitted in comments, string literals, character constants, and header names.

— The language supports wide-character constants and strings.

— The library has five basic functions that convert between multibyte and wide characters.

However, these five functions are often too restrictive and too primitive to develop portable international programs that manage characters. Consider a simple program that wants to count the number of characters, not bytes, in its input. The prototypical program,

```
#include <stdio.h>

int main(void)
    {
    int c, n = 0;

    while ((c = getchar()) != EOF)
        n++;
    printf("Count = %d\n", n);
    return 0;
    }
```

does not work as expected if the input contains multibyte characters; it always counts the number of bytes. It is certainly possible to rewrite this program using just some of the five basic conversion functions, but the simplicity and elegance of the above are lost.

ISO/IEC 9899:1990 deliberately chose not to invent a more complete multibyte and wide-character library, choosing instead to await their natural development as the C community acquired more experience with wide characters. The task of committee ISO JTC1/SC22/WG14 was to study the various existing implementations and, with care, develop this first amendment to ISO/IEC 9899:1990. The set of developed library functions is commonly called the *MSE* (Multibyte Support Extension).

Similarly, ISO/IEC 9899:1990 deliberately chose not to address in detail the problem of writing C source code with character sets such as the national variants of ISO 646. These variants often redefine several of the punctuation characters used to write a number of C tokens. The (admittedly partial) solution adopted was to add *trigraphs* to the language. Thus, for example, `??<` can appear anywhere in a C program that `{` can appear, even within a character constant or a string literal.

This amendment reponds to an international sentiment that more readable alternatives should also be provided, wherever possible. Thus, it adds to the language alternate spellings of several tokens. It also adds a library header, `<iso646.h>`, that defines a number of macros that expand to still other tokens which are less readable when spelled with trigraphs. Note, however, that trigraphs are still the only alternative to writing certain characters within a character constant or a string literal.

An important goal of any amendment to an international standard is to minimize *quiet changes* — changes in the definition of a programming language that transform a previously valid program into another valid program, or into an invalid program that need not generate a diagnostic message, with different behavior. (By contrast, changes that invalidate a previously valid program are generally considered palatable if they generate an obligatory diagnostic message at translation time.) Nevertheless, this amendment knowingly introduces two classes of quiet changes:

new tokens — The tokens `%:` and `%:%:` are just preprocessing tokens in ISO/IEC 9899:1990 but are given specific meanings in this amendment. An existing program that uses either of these tokens in a macro argument can behave differently as a result of this amendment.

new function names — Several names (with external linkage) not reserved to the implementation in ISO/IEC 9899:1990, such as `btowc`, are now so reserved if *any* translation unit in the program includes either of the headers `<wctype.h>` or `<wchar.h>`. An existing program that uses any of these names can behave differently as a result of this amendment.

B.2 Programming model based on wide characters

Using the MSE functions, a multibyte-character handling program can be written as easily and in the same style as a traditional single-byte based program. A programming model based on MSE function is as follows: First, a multibyte character or a multibyte string is read from an external file into a `wchar_t` object or a `wchar_t` array object by the `fgetwc` function, or another input functions based on the `fgetwc` function such as `getwchar`, `getwc`, or `fgetws`. During this read operation, a code conversion occurs — the input function converts the multibyte character to the corresponding wide character as if by a call to the `mbtowc` function.

After all necessary multibyte characters are read and converted, the `wchar_t` objects are processed in memory by the MSE functions, such as `iswxxx`, `wcstod`, `wcscpy`, `wmemcmp`, and so on. Finally, the resulting `wchar_t` objects are written to an external file as a sequence of multibyte characters by the `fputwc` function or other output functions based on the `fputwc` function, such as `putwchar`, `putwc`, or `fputws`. During this write operation, a code conversion occurs — the output function converts the wide character to the corresponding multibyte character as if by a call to the `wctomb` function.

In the case of the formatted input/output functions, a similar programming style can be applied, except that the character code conversion may also be done through extended conversion specifiers, such as `%ls` and `%lc`. For example, the wide-character based program corresponding to that shown in subclause B.1 can be written as follows;

```
#include <stdio.h>
#include <wchar.h>

int main(void)
    {
    wint_t wc;
    int n = 0;

    while ((wc = getwchar()) != WEOF)
        n++;
    wprintf(L"Count = %d\n", n);
    return 0;
    }
```

B.3 Parallelism versus improvement

When defining the MSE library functions, the committee could have chosen a design policy based either on *parallelism* or on *improvement*. "Parallelism" means that a function interface defined in this amendment is similar to the corresponding single-byte function in ISO/IEC 9899:1990. The number of parameters in corresponding functions are exactly same, and the types of parameters and the types of return values have a simple correspondence:

```
char <==> wchar_t        int  <==> wint_t
```

An approach using this policy is relatively easy.

On other hand, "improvement" means that a function interface in this amendment is changed from the corresponding single-byte functions in ISO/IEC 9899:1990 in order to resolve problems potentially contained in the existing functions. Or, a corresponding function is not introduced in this amendment when the functionality can be better attained through other functions. In an attempt to achieve improvement, there were numerous collisions of viewpoints on how to get the most appropriate interface. Moreover, much careful consideration and discussion among various experts in this area was necessary to decide which policy should be taken for each function. The current amendment is the result of this process.

The following is a list of the functions that manipulate characters in parallel:

ISO/IEC 9899:1990	Amendment
isalnum	iswalnum
isalpha	iswalpha
iscntrl	iswcntrl
isdigit	iswdigit
isgraph	iswgraph
islower	iswlower
isprint	iswprint
ispunct	iswpunct
isspace	iswspace
isupper	iswupper
isxdigit	iswxdigit
tolower	towlower
toupper	towupper
fprintf	fwprintf
fscanf	fwscanf
printf	wprintf
scanf	wscanf
sprintf	swprintf
sscanf	swscanf
vfprintf	vwfprintf
vprintf	vwprintf
vsprintf	vswprintf
fgetc	fgetwc
fgets	fgetws
fputc	fputwc
fputs	fputws
getc	getwc
getchar	getwchar
putc	putwc
putchar	putwchar
ungetc	ungetwc
strtod	wcstod
strtol	wcstol
strtoul	wcstoul
memcpy	wmemcpy
memmove	wmemmove
strcpy	wcscpy

`strncpy`	`wcsncpy`
`strcat`	`wcscat`
`strncat`	`wcsncat`
`memcmp`	`wmemcmp`
`strcmp`	`wcscmp`
`strcoll`	`wcscoll`
`strncmp`	`wcsncmp`
`strxfrm`	`wcsxfrm`
`memchr`	`wmemchr`
`strchr`	`wcschr`
`strcspn`	`wcscspn`
`strpbrk`	`wcspbrk`
`strrchr`	`wcsrchr`
`strspn`	`wcsspn`
`strstr`	`wcsstr`
`memset`	`wmemset`
`strlen`	`wcslen`
`strftime`	`wcsftime`

The following functions have different interfaces between single-byte and wide-character versions:

— Members of the `sprintf` family based on wide characters all have an extra `size_t` parameter, in order to repair the security hole that the existing functions carry. Compare:

```
int sprintf(char *s, const char *format, ...);
int swprintf(wchar_t *s, size_t n, const wchar_t *format, ...);

int vsprintf(char *s, const char *format, va_list arg);
int vswprintf(wchar_t *s, size_t n, const wchar_t *format,
    va_list arg);
```

— `wcstok`, the wide-character version of `strtok`, has an extra `wchar_t **` parameter, in order to eliminate the internal memory that the `strtok` function has to maintain. Compare:

```
char *strtok(char *s1, const char *s2);
wchar_t * wcstok(wchar_t * s1, const wchar_t *s2, wchar_t **ptr);
```

The following is a list of the functions in ISO/IEC 9899:1990 that do not have corresponding partners in the amendment for any of several reasons, such as redundancy, dangerous behavior, or a lack of need in a wide-character based program. Most of these can be rather directly replaced by other functions:

`gets`
`puts`
`perror`
`atof`
`atoi`
`atol`
`strerror`

Finally, the following is a list of the functions in this amendment that do not have corresponding partners in ISO/IEC 9899:1990. They were introduced to achieve better control over the conversion between multibyte characters and wide character, or to give character handling programs greater flexibility and simplicity:

`wctype`
`iswctype`
`wctrans`
`towctrans`
`fwide`
`btowc`
`wctob`
`mbsinit`
`mbrlen`
`mbrtowc`
`wcrtomb`
`mbsrtowcs`
`wcsrtombs`

B.4 Support for invariant ISO 646

With its rich set of operators and punctuators, the C language makes heavy demands on the ASCII character set. Even before the language was standardized, it presented problems to those who would move C to EBCDIC machines. More than one vendor provided alternate spellings for some of the tokens that used characters with no EBCDIC equivalent. With the spread of C throughout the world, such representation problems have only grown worse.

ISO 646, the international standard corresponding to ASCII, permits national variants of a number of the characters used by C. Strictly speaking, this is not a problem in representing C programs, since the necessary characters exist in all such variants. They just print funny. Displaying C programs for human edification suffers, however, since the operators and punctuators can be hard to recognize in their various altered forms.

ISO/IEC 9899:1990 addresses the problem in a different way. It provides replacements at the level of individual characters using three-character sequences called *trigraphs*. For example, `??<` is entirely equivalent to `{`, even within a character constant or string literal. While this approach provides a complete solution for the known limitations of EBCDIC and ISO 646, the result is arguably not highly readable.

Thus, this amendment provides a set of more readable *digraphs*. These are two-character alternate spellings for several of the operators and punctuators that can be hard to read with ISO 646 national variants. Trigraphs are still required within character constants and string literals, but at least the commoner operators and punctuators can have more suggestive spellings using digraphs.

The added digraphs were intentionally kept to a minimum. Wherever possible, the committee instead provided alternate spellings for operators in the form of macros defined in the new header **<iso646.h>**. Alternate spellings are provided for the preprocessing operators # and ## because they cannot be replaced by macro names. Digraphs are also provided for the punctuators `[`, `]`, `{`, and `}` because macro names proved to be a less readable alternative. The committee recognizes that the solution offered in this amendment is incomplete and involves a mixture of approaches, but nevertheless believes that it can help make Standard C programs more readable.

B.5 Headers

B.5.1 <wchar.h>

B.5.1.1 Prototypes in <wchar.h>

Function prototypes for the MSE library functions had to be included in some header. The Committee considered following ideas:

1. Introduce new headers such as **<wctype.h>**, **<wstdio.h>**, and **<wstring.h>**, corresponding to the existing headers specified in ISO/IEC 9899:1990, such as **<ctype.h>**, **<stdio.h>**, and **<string.h>**.

2. Declare all the MSE function prototypes in **<stdlib.h>**, where **wchar_t** is already defined.

3. Introduce a new header and declare all the MSE function prototypes in the new header.

4. Declare the MSE function prototypes in existing headers specified in ISO/IEC 9899:1990 related these functions

The drawback to idea 1 is that the relationship between new headers and existing ones, especially their dependencies, becomes complicated. For example, two headers may have to be included prior to including **<wstdio.h>**, as in:

```
#include <stdlib.h>
#include <stdio.h>
#include <wstdio.h
```

The drawback to idea 2 is that the program has to include many prototype declarations even if the program does not need declarations in **<stdlib.h>** other than existing ones. And the committee strongly opposed adding any identifiers to existing headers.

The drawback to idea 3 is that it introduces an asymmetry between existing headers and new header.

The drawback to idea 4 is that the committee strongly opposed adding any identifiers to existing headers.

So the committee decided to introduce a new header **<wchar.h>** as the least objectionable way to declare all MSE function prototypes. (Later, the committee split off the functions analogous to those in **<ctype.h>** and placed their declarations in the new header **<wctype.h>**, as described in subclause B.5.2.)

B.5.1.2 Types and macros in <wchar.h>

The committee was concerned that the definitions of types and macros in **<wchar.h>** be specified efficiently. One goal was to require that only the header **<wchar.h>** need be included to use the MSE library functions. But there were strong objections to declaring existing types such as **FILE** in the new header.

The definitions in **<wchar.h>** are thus limited to those types and macros that are largely independent of the existing library. The existing header **<stdio.h>** must also be included along with **<wchar.h>** when the program needs explicit definitions of either of the types **FILE** and **fpos_t**. (How these types are defined in **<stdio.h>** may need to be revised so that suitable synonyms, with reserved names, can be used within **<wchar.h>**.)

B.5.2 <wctype.h>

The committee originally intended to place all MSE functionality in a single header, **<wchar.h>**, as explained in subclause B.5.1.1. It found, however, that this header was excessively large, even compared to the existing large headers **<stdio.h>** and **<stdlib.h>**. The committee also observed that the wide-character classification and mapping functions seemed to form a separate group. (These are functions that typically have names of the form ***iswxxx*** or ***towxxx***.) A translation unit could well make use of most of the functionality of the MSE without using this separate group. Equally, a translation unit might need the wide-character classification and mapping functions without needing the other MSE functions.

Hence, the committee decided to form a separate header, **<wctype.h>**, closely analogous to the existing **<ctype.h>**. That division also reduced the size of **<wchar.h>** to more manageable proportions.

B.6 Wide-character classification functions

Eleven ***iswxxx*** functions have been introduced to correspond to the character-testing functions defined in ISO/IEC 9899:1990. Each wide-character testing function is specified in parallel with the matching single-byte character handling function. However, the following changes were also introduced.

B.6.1 Locale dependency of *iswxxx* functions

The behavior of character-testing functions in ISO/IEC 9899:1990 is affected by the current locale. And some of the functions have implementation-defined aspects only when not in the **"C"** locale. For example, in the **"C"** locale, **islower** returns true (nonzero) only for lower-case letters (as defined in subclause 5.2.1 of ISO/IEC 9899:1990).

This existing **"C"** locale restriction for character testing functions in ISO/IEC 9899:1990 has been replaced with a supersetting constraint for wide-character testing functions. There is no special description of **"C"** locale behavior for the ***iswxxx*** functions. Instead, the following rule is applied to any locale. When a character **c** causes ***isxxx(c)*** to return true, the corresponding wide character **wc** shall cause the corresponding function call ***iswxxx*****(wc)** to return true.

```
isxxx(c) != 0 ==> iswxxx(wc) !=0
```

B.6.2 Changed space-character handling

The space character (**' '**) is treated specially in **isprint**, **isgraph**, and **ispunct**. Space-character handling in the corresponding wide-character functions differs from that specified in ISO/IEC 9899:1990. The corresponding wide-character functions return true for all wide characters for which **iswspace** returns true, instead of just the single space character. Therefore, the behaviors of the **iswgraph** and **iswpunct** functions may differ from their matching functions in ISO/IEC 9899:1990 in this regard. (See the footnote concerning **iswgraph** in this amendment).

B.7 Extensible classification and mapping functions

There are eleven standard character-testing functions defined in ISO/IEC 9899:1990. As the number of supported locales increases, the requirements for additional character classifications grows, and varies from locale to locale. To satisfy this requirement, many existing implementations, especially for non-English speaking countries, have been defining new ***isxxx*** functions, such as **iskanji**, **ishanzi**, etc.

However, this approach adds to the global namespace problem and is not flexible at all in supporting additional classification requirements. Therefore, in this amendment, a pair of extensible wide-character classification functions, **wctype** and **iswctype**, are introduced to satisfy the open-ended requirements for character classification. Since the name of a character classification is passed as an argument to the **wctype** function, it does not add to problem of global namespace pollution. And these generic interfaces allow a program to test if the classification is available in the current locale, and to test for locale-specific character classifications, such as **kanji** or **hiragana** in Japanese.

In the same way, a pair of wide-character mapping functions, **wctrans** and **towctrans**, are introduced to support locale-specific character mappings. One of the example of applying this functionality is the mappings between **hiragana** and **katakana** in a Japanese character set.

B.8 Generalized multibyte characters

ISO/IEC 9899:1990 intentionally restricted the class of acceptable encodings for multibyte characters. One goal was to ensure that, at least in the initial shift state, the characters in the basic C character set have multibyte representations that are single characters with the same code as the single-byte representation. The other was to ensure that the null byte should never appear as the second or subsequent byte of any multibyte code. Hence, `'a'` is always `'a'` (at least initially) and `'\0'` is always `'\0'`, to put matters most simply.

While these may be reasonable restrictions within a C program, they hamper the ability of the MSE functions to read arbitrary wide-oriented files. For example, a system may wish to represent files as sequences of ISO 10646 characters. Reading or writing such a file as a wide-oriented stream should be an easy matter. At most, the library may have to map between native and some canonical byte order in the file. In fact, it is easy to think of an ISO 10646 file as being some form of multibyte file — except that it violates both restrictions described above. (The code for `'a'` can look like the four-byte sequence `"\0\0\0a"`, for example.)

Thus, the MSE introduces the notion of a *generalized multibyte encoding*. It subsumes all the ways the committee can currently imagine that operating systems will represent files containing characters from a large character set. (Such encodings are valid only in files — they are still not permitted as internal multibyte encidings.)

B.9 Streams and files

B.9.1 Conversion state

It is necessary to convert between multibyte characters and wide characters within wide-character input/output functions. The conversion state, introduced in subclause 4.5.3.2 of this amendment, is used to help perform this conversion. Every wide-character input/output function makes use of (and updates) the conversion state held in the **FILE** object controlling the wide-oriented stream.

The conversion state in the **FILE** object augments the file position within the corresponding multibyte character stream with the parse state for the next multibyte character to obtain from that stream. For state-dependent encodings, the remembered shift state is a part of this parse state, and hence a part of the conversion state. (Note that a multibyte encoding that has *any* characters requiring two or more bytes needs a nontrivial conversion state, even if it is not a state-dependent encoding.)

The wide-character input/output functions behave as if:

— a **FILE** object includes a hidden **mbstate_t** object;

— the wide-character input/output functions use this hidden object as the state argument to the **mbrtowc** or **wcrtomb** functions that perform the conversion between multibyte characters in the file and wide characters inside the program.

B.9.2 Implementation

The committee assumed that only wide-character input/output functions can maintain consistency between the conversion-state information and the stream. The byte input/output functions do nothing with the conversion state information in the **FILE** object. The wide-character input/output functions are designed on the premise that they always begin executing with the stream positioned at the boundary between two multibyte characters.

The committee felt that it would be intolerable to require implementors to implement these functions without such a guarantee. Since executing a byte input/output function on a wide-oriented stream may well leave the file position indicator at other than the boundary between two multibyte characters, the committee decided to prohibit such use of the byte input/output functions.

B.9.2.1 Seek operations

An `fpos_t` object for a stream in a state-dependent encoding includes the shift state information for the corresponding stream. In order to ensure the behavior of subsequent wide-character input/output functions in a state-dependent encoding environment, a seek operation should reset the conversion state corresponding to the file position as well as restoring the file position.

The traditional seek functions `fseek` and `ftell` may not be adequate in such an environment, because a `long` object may be too small to hold both the conversion state information and the file position indicator. Thus, the newer `fsetpos` and `fgetpos` are preferred, since they can store as much information as necessary in an `fpos_t` object.

B.9.2.2 State-dependent encodings

With state-dependent encodings, a `FILE` object must include the conversion state for the stream. The committee felt strongly that programmers should not have to handle the tedious details of keeping track of conversion states for wide-character input/output. There is no means, however, for programmers to access the internal shift state or conversion state in a `FILE` object.

B.9.2.3 Multiple encoding environments

A *multiple encoding environment* has two or more different encoding schemes for files. In such an environment, some programmers will want to handle two or more multibyte character encodings on a single platform, possibly within a single program. There is, for example, an environment in Japan that has two or more encoding rules for a single character set. Most implementations for Japanese environments should provide for such multiple encodings.

During program execution, the wide-character input/output functions get information about the current encodings from the `LC_CTYPE` category of the current locale. When writing a program for a multiple encoding environment, the programmer should be aware of the proper `LC_CTYPE` category for each opened file. During every access to a file, the appropriate `LC_CTYPE` category should be restored.

The encoding-rule information is effectively a part of the conversion state. Thus, the encoding-rule information should be stored with the hidden `mbstate_t` object within the `FILE` object. (Some implementations may even choose to store the encoding rule as part of the value of an `fpos_t` object.)

The conversion state just created when a file is opened is said to have *unbound* state because it has no relations to any of the encoding rules. Just after the first wide-character input/output operation, the conversion state is *bound* to the encoding rule which corresponds to the `LC_CTYPE` category of the current locale. The following is a summary of the relations between various objects, the shift state, and the encoding rules:

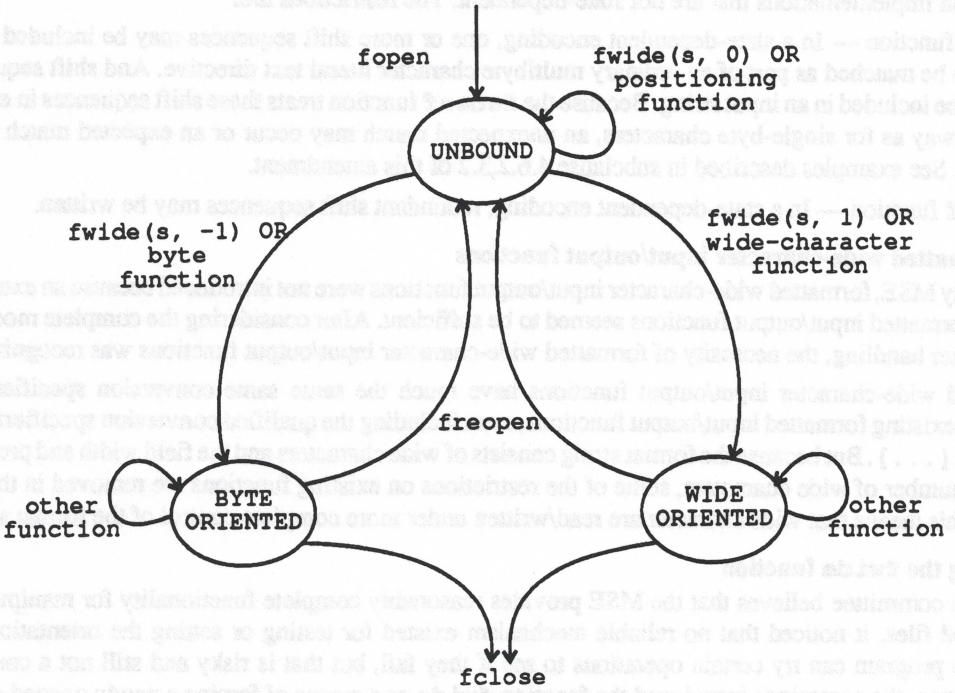
		fpos_t	FILE
shift state		included	included
encoding rule		maybe	included
changing LC_CTYPE	(unbound)	no effect	affected
	(bound)	no effect	no effect

B.9.3 Byte versus wide-character input/output

Both the wide-character input/output functions and the byte input/output functions refer the same type of object (a `FILE` object). As described in subclause B.9.2, however, there is a constraint on mixed usage of the two type of input/output functions. That is, if a wide-character input/output functions is executed for a `FILE` object, its stream becomes wide-oriented and no byte input/output functions shall be applied later (and conversely).

The reason for this constraint is to ensure consistency between the current file position and the current conversion state in the `FILE` object. Executing one of the byte input/output functions for a wide-oriented stream breaks this consistency, because the byte input/output functions may (or should) ignore the conversion state information in the `FILE` object.

The following diagram shows the state transitions of a stream in response to various input/output functions.



B.9.4 Text versus binary input/output

In some implementations, such as UNIX, there are streams which look the same whether read or written as text or binary. (For example, arbitrary file-positioning operations are supported even in text mode.) In such an implementation, the committee specifies the following usage of the wide-character input/output functions. A file opened as a binary stream should obey the usage constraints of placed upon text streams when accessed as a wide-character stream. (For example, file positioning is more restricted.)

So an implementation of the wide-character input/output functions can rely on the premise that programmers use the wide-character input/output functions with a binary stream under the same constraints as for a text stream. An implementation may also provide wide-character input/output functions that behave correctly on an unconstrained binary stream. However, the behavior of the wide-character input/output functions on such an unconstrained binary stream cannot be ensured by all implementations.

B.10 Formatted input/output functions

B.10.1 Enhancing existing formatted input/output functions

The simplest extension for wide-character input/output is to use existing formatted input/output functions with existing (byte-oriented) streams. In this case, data consists of characters only (such as strings) are treated as sequences of wide character and other data (such as numerical values) are treated as sequences of single-byte characters. Though this is not a complete model for wide-character processing, it is a common extension among some existing implementations in Japan. So the committee decided to include a similar extension.

At first, the new conversion specifiers `%S` and `%C` were added to the existing formatted input and output functions, to handle a wide-character string and a wide character respectively. After long discussions about the actual implementation and future library directions (in subclause 7.13.6 of ISO/IEC 9899:1990), these specifiers were withdrawn. They were replaced with the qualified conversion specifiers `%ls` and `%lc` (with the addition of `%l[...]` in the formatted input functions). Note that even though the new qualifier is introduced as an extension for wide-character processing, the field width and the precision still specify the number of bytes (in the multibyte representation in the stream).

To implement these new conversion specifiers efficiently, a new set of functions is required. These parse or generate multibyte sequences "restartably." Thus, the functions described in subclauses 4.6.5.1, 4.6.5.2, 4.6.5.3, and 4.6.5.4 of this amendment were introduced.

Because these new conversions are pure extensions to ISO/IEC 9899:1990, they have several essential restriction on their ability to deal with state-dependent encodings. It is expected, therefore, that they will be most useful in implementations that are not state-dependent. The restrictions are:

— **`fscanf`** function — In a state-dependent encoding, one or more shift sequences may be included in the format, to be matched as part of an ordinary multibyte character literal text directive. And shift sequences may also be included in an input string. Because the **`fscanf`** function treats these shift sequences in exactly the same way as for single-byte characters, an unexpected match may occur or an expected match might not occur. See examples described in subclause 4.6.2.3.2 of this amendment.

— **`fprintf`** function — In a state-dependent encoding, redundant shift sequences may be written.

B.10.2 Formatted wide-character input/output functions

In the early MSE, formatted wide-character input/output functions were not introduced because an extension to existing formatted input/output functions seemed to be sufficient. After considering the complete model for wide-character handling, the necessity of formatted wide-character input/output functions was recognized.

Formatted wide-character input/output functions have much the same same conversion specifiers and qualifiers as existing formatted input/output functions, even including the qualified conversion specifiers **`%lc`**, **`%ls`**, and **`%l[...]`**. But because the format string consists of wide characters and the field width and precision specify the number of wide characters, some of the restrictions on existing functions are removed in the new functions. This means that wide character are read/written under more complete control of the format string.

B.11 Adding the `fwide` function

While the committee believes that the MSE provides reasonably complete functionality for manipulating wide-oriented files, it noticed that no reliable mechanism existed for testing or setting the orientation of a stream. (The program can try certain operations to see if they fail, but that is risky and still not a complete strategy.) Hence, the committee introduced the function **`fwide`** as a means of forcing a newly opened stream into the desired orientation without attempting any input/output on the stream. The function also serves as a passive means of testing the orientation of a stream, either before or after the orientation has been fixed. And it serves as a way to bind an encoding rule to a wide-oriented stream under more controlled circumstances. (See subclause B.9.2.3.)

B.12 Single-byte wide-character conversion functions

Two single-byte wide-character conversion functions, **`btowc`** and **`wctob`**, have been introduced in this amendment. These functions simplify mappings between a single-byte character and its corresponding wide character (if any).

ISO/IEC 9899:1990 specifies the rule **`L'x' == 'x'`** for a member **`x`** of the basic character set. The committee discussed whether to relax or tighten this rule. In this amendment, this rule is preserved without any changes. Applying the rule to all single-byte characters, however, imposes an unnecessary constraint on implementation with regard to wide-character encodings. It prohibits an implementation from having a common wide-character encoding for multiple multibyte encodings.

On the other hand, relaxing or removing the rule was considered to be inappropriate in terms of practical implementations. The new function **`wctob`** can be used to test safely and quickly whether a wide character corresponds to some single-byte character. For example, when the format string on a **`scanf`** function call is parsed and searched for a white space character, the **`wctob`** function can be used in conjunction with the **`isspace`** function. (See the specification of the ***`iswxxx`*** functions in subclause 4.5.2.1 of this amendment.)

Similarly, there are frequent occasions in wide-character processing code, especially in the wide-character handling library functions, where it is necessary to test quickly and efficiently whether a single-byte character is the first and only character of a valid multibyte character. This is the reason for introducing the **`btowc`** function. Note that, for some encodings, **`btowc`** can be reduced to a simple in-line expression.

B.13 Extended conversion utilities

Although ISO/IEC 9899:1990 allows multibyte characters to have state-dependent encoding (subclause 5.2.1.2), the original functions are not always sufficient to efficiently support state-dependent encodings, due to the following limitations of the multibyte character conversion functions (subclause 7.10.7):

1. Since the functions maintain shift state information internally, they cannot handle multiple strings at the same time.

2. The formatted output functions may write redundant shift sequences, and the formatted input functions cannot reliably parse input with arbitrary or redundant shift sequences.

3. The multibyte-string conversion functions (subclause 7.10.8) have an inconvenient shortcoming, regardless of state dependency of the encoding. When an encoding error occurs, these functions return –1 without any information on the location where the conversion stopped.

For all these reasons, the committee felt it necessary to augment the set of conversion functions in this amendment.

B.13.1 Conversion state

To handle multiple strings with a state-dependent encoding, the committee introduced the concept of conversion state. The conversion state determines the behavior of a conversion between multibyte and wide-character encodings. For conversion from multibyte to wide character, the conversion state stores information such as the position within the current multibyte character (as a sequence of characters or a wide-character accumulator). And for conversions in either direction, the conversion state stores the current shift state (if any) and possibly the encoding rule.

The non-array object type **`mbstate_t`** is defined to encode the conversion state. A zero-valued **`mbstate_t`** object is assumed to describe the initial conversion state. (It is not necessarily the *only* way to encode the initial conversion state, however.) Before any operations are performed on it, such a zero-valued object is *unbound*. Once a multibyte or wide-character conversion function executes with the **`mbstate_t`** object as an argument, however, the object becomes *bound* and holds the above information.

The conversion functions maintain the conversion state in an **`mbstate_t`** object according to the encoding specified in the **`LC_CTYPE`** category of the current locale. Once the conversion starts, the functions will work as if the encoding scheme were not changed provided all three of the following conditions obtain:

— the function is applied to the same string as when the **`mbstate_t`** object was first bound;

— the **`LC_CTYPE`** category setting is the same as when the **`mbstate_t`** object was first bound;

— the conversion direction (multibyte to wide character, or wide character to multibyte) is the same as when the **`mbstate_t`** object was first bound.

B.13.2 Conversion utilities

Once the **`mbstate_t`** object was introduced, the committee discussed the need for additional functions to manipulate such objects.

B.13.2.1 Initializing conversion states

Though a method to initialize the object is needed, the committee decided that it would be better not to define too many functions in this amendment. Thus the committee decided to specify only one way to make an **`mbstate_t`** object represent the initial conversion state — by initializing it with zero. No initializing function is supplied.

B.13.2.2 Comparing conversion states

The committee reached the conclusion that it may be impossible to define the equality between two conversion states. If two **`mbstate_t`** objects have the same values for all attributes, they mignt be the same. However, they might also have different values and still represent the same conversion state. No comparison function is supplied.

Testing for initial shift state

The function `mbsinit` was added to test whether an `mbstate_t` object describes the initial conversion state or not, because this state does not always correspond to a certain set of component values (and the components cannot be portably compared anyway). The function is necessary because many functions in the amendment treat the initial shift state as a special condition.

Regarding problems 2 and 3 described at the beginning of subclause B.13, the committee introduced a method to distinguish between an invalid sequence of bytes and a valid prefix to a still incomplete multibyte character. When encountering such an incomplete multibyte sequence, the `mbrlen` and `mbrtowc` functions return –2 instead of –1, and the character accumulator in the `mbstate_t` object stores the partial character information. Thus, the user can resume the pending conversion later, and can even convert a sequence one byte at a time.

The new multibyte/wide-string conversion utilities are thus made *restartable* by using the character accumulator and shift-state information stored in an `mbstate_t` object argument. As part of this enhancement, the functions also have a parameter of type pointer to pointer to the source string. The function uses this argument to store a pointer to the position where the conversion stopped.

B.14 Column width

The number of characters to be read or written can be specified in existing formatted input/output functions. On a traditional display device that displays characters with fixed pitch, the number of characters is directly proportional to the width occupied by these characters. So the display format can be specified through the field width and/or the precision.

In formatted wide-character input/output functions, the field width and the precision specifies the number of wide characters to be read or written. The number of wide characters is not always directly proportional to the width of their display. For example, with Japanese traditional display devices, a single-byte character such as an ASCII character has half the width of a Kanji character, even though each of them is treated as one wide character. To control the display format for wide characters, a set of formatted wide-character input/output functions were proposed whose metric was the column width instead of the character count.

This proposal was supported only by Japan. Critics observed that the proposal was based on such traditional display devices with a fixed width of characters, while many modern display devices support a broad assortment of proportional pitch type faces. Hence, it was questioned whether the extra input/output functions in this proposal were really needed or were sufficiently general. Also considered were another set of functions that return the column width for any kind of display devices for a given wide character or wide-character string; but these seemed to be beyond the scope of C language. Thus all proposals regarding column width were withdrew.

If an implementor needs this kind of functionality, there are a few ways to extend wide-character output functions and still remain conforming to this amendment. For example, the field width prefixed with a # can specify the column width as shown below:

`%#N` — set the counting mode to "printing positions" and reset the `%n` counter;

`%N` — set the counting mode back to "wide characters" and reset the `%n` counter.

Index

%: operator, 3.1
%:%: operator, 3.1
%: punctuator, 3.2
%> punctuator, 3.2
:> operator, 3.1
:> punctuator, 3.2
<% punctuator, 3.2
<: operator, 3.1
<: punctuator, 3.2
and macro, 4.4
and_eq macro, 4.4
bitand macro, 4.4
bitor macro, 4.4
btowc function, 4.6.5.1.1
byte and character problem, clause 1
byte input/output functions, 4.6.2
byte-oriented streams, 4.6.2.1
compl macro, 4.4
control wide character, 4.5.2
conversion state, 4.6.5
EILSEQ macro, 4.3, 4.6.2.2
encoding error, 4.6.2.2, 4.6.2.5.1, 4.6.2.5.5
fgetwc function, 4.6.2.5.1
fgetws function, 4.6.2.5.2
fprintf function, 4.6.2.3.1
fputwc function, 4.6.2.5.3
fputws function, 4.6.2.5.4
fscanf function, 4.6.2.3.2
fwide function, 4.6.2.5.10
fwprintf function, 4.6.2.4.1
fwscanf function, 4.6.2.4.2
getwc function, 4.6.2.5.5
getwchar function, 4.6.2.5.6
iso646.h header, 4.4, clause 2
iswalnum function, 4.5.2.1.1
iswalpha function, 4.5.2.1.2
iswcntrl function, 4.5.2.1.3
iswctype function, 4.5.2.2.1
iswdigit function, 4.5.2.1.4
iswgraph function, 4.5.2.1.5
iswlower function, 4.5.2.1.6
iswprint function, 4.5.2.1.7
iswpunct function, 4.5.2.1.8
iswspace function, 4.5.2.1.9
iswupper function, 4.5.2.1.10
iswxdigit function, 4.5.2.1.11
mbrlen function, 4.6.5.3.1
mbrtowc function, 4.6.5.3.2
mbsinit function, 4.6.5.2
mbsrtowcs function, 4.6.5.4.1
mbstate_t type, 4.6.1
not macro, 4.4
not_eq macro, 4.4
null wide character, 4.1
or macro, 4.4
or_eq macro, 4.4
orientation, stream, 4.6.2.1
printing wide character, 4.6.2
putwc function, 4.6.2.5.7
putwchar function, 4.6.2.5.8
shift sequence, 4.1
size_t type, 4.6.1
stream orientation, 4.6.2.1
swprintf function, 4.6.2.4.5
swscanf function, 4.6.2.4.6
towctrans, 4.5.3.2.2
towlower function, 4.5.3.1.1
towupper function, 4.5.3.1.2
ungetwc function, 4.6.2.5.9
vfwprintf function, 4.6.2.4.7
vwprintf function, 4.6.2.4.8
vswprintf function, 4.6.2.4.9
WCHAR_MAX macro, 4.6.1
WCHAR_MIN macro, 4.6.1
wchar_t type, 4.6.1
wcrtomb function, 4.6.5.3.3
wcscat function, 4.6.3.3.1
wcschr function, 4.6.3.5.1
wcscmp function, 4.6.3.4.1
wcscoll function, 4.6.3.4.2
wcscpy function, 4.6.3.2.1
wcscspn function, 4.6.3.5.2
wcsftime function, 4.6.4
wcslen function, 4.6.3.5.8
wcsncat function, 4.6.3.3.2
wcsncmp function, 4.6.3.4.3
wcsncpy function, 4.6.3.2.2
wcspbrk function, 4.6.3.6.2
wcsrchr function, 4.6.3.5.4
wcsrtombs function, 4.6.5.4.2
wcsspn function, 4.6.3.5.5
wcsstr function, 4.6.3.5.6
wcstod function, 4.6.3.1.1
wcstok function, 4.6.3.5.7
wcstol function, 4.6.3.1.2
wcstoul function, 4.6.3.1.3
wcsxfrm function, 4.6.3.4.4
wctob function, 4.6.5.1.2
wctrans function, 4.5.3.2.1
wctrans_t type, 4.5.1
wctype function, 4.5.2.2.1
wctype_t type, 4.5.1
WEOF macro, 4.5.1, 4.6.1
wide-character, 4.1
wide-character input functions, 4.6.2
wide-character input/output functions, 4.6.2
wide-character output functions, 4.6.2
wide string, 4.1
wide-oriented streams, 4.6.2.1
wint_t type, 4.5.1, 4.6.1
wmemchr function, 4.6.3.6,1
wmemcmp function, 4.6.3.6.2
wmemcpy function, 4.6.3.6.3
wmemmove function, 4.6.3.6.4
wmemset function, 4.6.3.6.5
wprintf function, 4.6.2.4.3
wscanf function, 4.6.2.4.4
xor macro, 4.4
xor_eq macro, 4.4