Document Number: P1720R2
Date: 2019-11-07
Reply to: Marshall Clow
mclow.lists@gmail.com

Mandating the Standard Library: Clause 28 - Localization library

With the adoption of P0788R3, we have a new way of specifying requirements for the library clauses of the standard. This is one of a series of papers reformulating the requirements into the new format. This effort was strongly influenced by the informational paper P1369R0.

The changes in this series of papers fall into three broad categories.

— Change "participate in overload resolution" wording into "Constraints" elements

— Change "Requires" elements into either "Mandates" or "Expects", depending (mostly) on whether or not they can be checked at compile time.

— Drive-by fixes (hopefully very few)

This paper covers Clause 28 (Localization), and is based on N4830.

The entire clause is reproduced here, but the changes are confined to a few sections:

— locale 28.3.1

— locale.cons 28.3.1.2

— locale.members 28.3.1.3

— locale.statics 28.3.1.5

— locale.global.templates 28.3.2

— facet.ctype.char.members 28.4.1.3.2

— locale.codecvt.virtuals 28.4.1.4.2

— facet.numpunct.virtuals 28.4.3.1.2

— locale.time.get.members 28.4.5.1.1

— locale.time.get.virtuals 28.4.5.1.2

— locale.messages 28.4.7.1

— locale.messages.members 28.4.7.1.1

— locale.messages.virtuals 28.4.7.1.2

Drive-by fixes:

— Reworked [facet.ctype.char.members]/2.

— Removed several meaningless paragraphs in [locale.cons].

— Removed a 'shall' from [locale.statics]

— Changed several `basic_string<char>` types to just `string`.

— Added 'is' in a few places to make english sentences.

Changes from R0

— Updated to N4830

— Reworked the change in [facet.ctype.char.members]/2 to use 'is a valid range'

— Removed a 'shall' from [locale.statics]

— Incorporated feedback from LWG teleconference 6-Sep-2019.

Changes from R1

— Restored [locale.cons] p16 and 17 based on LWG feedback

— made sure a `string` was in code font.

Thanks to Daniel Krügler for his advice and reviews.

Help for the editors: The changes here can be viewed as latex sources with the following commands

```
git clone git@github.com:mclow/mandate.git
cd mandate
git diff master..chapter28 locales.tex
```

28 Localization library [localization]

28.1 General [localization.general]

1 This Clause describes components that C++ programs may use to encapsulate (and therefore be more portable when confronting) cultural differences. The locale facility includes internationalization support for character classification and string collation, numeric, monetary, and date/time formatting and parsing, and message retrieval.

2 The following subclauses describe components for locales themselves, the standard facets, and facilities from the ISO C library, as summarized in Table 100.

Table 100: Localization library summary [tab:localization.summary]

	Subclause	Header
28.3	Locales	`<locale>`
28.4	Standard `locale` categories	
28.5	C library locales	`<clocale>`

28.2 Header `<locale>` synopsis [locale.syn]

```
namespace std {
  // 28.3.1, locale
  class locale;
  template<class Facet> const Facet& use_facet(const locale&);
  template<class Facet> bool         has_facet(const locale&) noexcept;

  // 28.3.3, convenience interfaces
  template<class charT> bool isspace (charT c, const locale& loc);
  template<class charT> bool isprint (charT c, const locale& loc);
  template<class charT> bool iscntrl (charT c, const locale& loc);
  template<class charT> bool isupper (charT c, const locale& loc);
  template<class charT> bool islower (charT c, const locale& loc);
  template<class charT> bool isalpha (charT c, const locale& loc);
  template<class charT> bool isdigit (charT c, const locale& loc);
  template<class charT> bool ispunct (charT c, const locale& loc);
  template<class charT> bool isxdigit(charT c, const locale& loc);
  template<class charT> bool isalnum (charT c, const locale& loc);
  template<class charT> bool isgraph (charT c, const locale& loc);
  template<class charT> bool isblank (charT c, const locale& loc);
  template<class charT> charT toupper(charT c, const locale& loc);
  template<class charT> charT tolower(charT c, const locale& loc);

  // 28.4.1, ctype
  class ctype_base;
  template<class charT> class ctype;
  template<>            class ctype<char>;     // specialization
  template<class charT> class ctype_byname;
  class codecvt_base;
  template<class internT, class externT, class stateT> class codecvt;
  template<class internT, class externT, class stateT> class codecvt_byname;

  // 28.4.2, numeric
  template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class num_get;
  template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class num_put;
  template<class charT>
    class numpunct;
```

```
  template<class charT>
    class numpunct_byname;

  // 28.4.4, collation
  template<class charT> class collate;
  template<class charT> class collate_byname;

  // 28.4.5, date and time
  class time_base;
  template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class time_get;
  template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class time_get_byname;
  template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class time_put;
  template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class time_put_byname;

  // 28.4.6, money
  class money_base;
  template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class money_get;
  template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class money_put;
  template<class charT, bool Intl = false>
    class moneypunct;
  template<class charT, bool Intl = false>
    class moneypunct_byname;

  // 28.4.7, message retrieval
  class messages_base;
  template<class charT> class messages;
  template<class charT> class messages_byname;
}
```

1 The header <locale> defines classes and declares functions that encapsulate and manipulate the information peculiar to a locale.[257]

28.3 Locales [locales]

28.3.1 Class locale [locale]

```
namespace std {
  class locale {
  public:
    // types
    class facet;
    class id;
    using category = int;
    static const category   // values assigned here are for exposition only
      none     = 0,
      collate  = 0x010, ctype    = 0x020,
      monetary = 0x040, numeric  = 0x080,
      time     = 0x100, messages = 0x200,
      all = collate | ctype | monetary | numeric | time  | messages;

    // construct/copy/destroy
    locale() noexcept;
    locale(const locale& other) noexcept;
    explicit locale(const char* std_name);
    explicit locale(const string& std_name);
    locale(const locale& other, const char* std_name, category);
    locale(const locale& other, const string& std_name, category);
    template<class Facet> locale(const locale& other, Facet* f);
```

257) In this subclause, the type name struct tm is an incomplete type that is defined in <ctime>.

```
    locale(const locale& other, const locale& one, category);
    ~locale();                  // not virtual
    const locale& operator=(const locale& other) noexcept;
    template<class Facet> locale combine(const locale& other) const;

    // locale operations
    basic_string<char>string name() const;

    bool operator==(const locale& other) const;

    template<class charT, class traits, class Allocator>
      bool operator()(const basic_string<charT, traits, Allocator>& s1,
                      const basic_string<charT, traits, Allocator>& s2) const;

    // global locale objects
    static       locale  global(const locale&);
    static const locale& classic();
  };
}
```

1 Class `locale` implements a type-safe polymorphic set of facets, indexed by facet *type*. In other words, a facet has a dual role: in one sense, it's just a class interface; at the same time, it's an index into a locale's set of facets.

2 Access to the facets of a `locale` is via two function templates, `use_facet<>` and `has_facet<>`.

3 [*Example*: An iostream `operator<<` might be implemented as:[258]

```
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<< (basic_ostream<charT, traits>& s, Date d) {
  typename basic_ostream<charT, traits>::sentry cerberos(s);
  if (cerberos) {
    ios_base::iostate err = ios_base::iostate::goodbit;
    tm tmbuf; d.extract(tmbuf);
    use_facet<time_put<charT, ostreambuf_iterator<charT, traits>>>(
      s.getloc()).put(s, s, s.fill(), err, &tmbuf, 'x');
    s.setstate(err);            // might throw
  }
  return s;
}
```

— *end example*]

4 In the call to `use_facet<Facet>(loc)`, the type argument chooses a facet, making available all members of the named type. If `Facet` is not present in a locale, it throws the standard exception `bad_cast`. A C++ program can check if a locale implements a particular facet with the function template `has_facet<Facet>()`. User-defined facets may be installed in a locale, and used identically as may standard facets.

5 [*Note*: All locale semantics are accessed via `use_facet<>` and `has_facet<>`, except that:

(5.1) — A member operator template `operator()(const basic_string<C, T, A>&, const basic_string<C, T, A>&)` is provided so that a locale may be used as a predicate argument to the standard collections, to collate strings.

(5.2) — Convenient global interfaces are provided for traditional `ctype` functions such as `isdigit()` and `isspace()`, so that given a locale object `loc` a C++ program can call `isspace(c, loc)`. (This eases upgrading existing extractors (**??**).)

— *end note*]

6 Once a facet reference is obtained from a locale object by calling `use_facet<>`, that reference remains usable, and the results from member functions of it may be cached and re-used, as long as some locale object refers to that facet.

7 In successive calls to a locale facet member function on a facet object installed in the same locale, the returned result shall be identical.

258) Note that in the call to `put` the stream is implicitly converted to an `ostreambuf_iterator<charT, traits>`.

8 A `locale` constructed from a name string (such as `"POSIX"`), or from parts of two named locales, has a name; all others do not. Named locales may be compared for equality; an unnamed locale is equal only to (copies of) itself. For an unnamed locale, `locale::name()` returns the string `"*"`.

9 Whether there is one global locale object for the entire program or one global locale object per thread is implementation-defined. Implementations should provide one global locale object per thread. If there is a single global locale object for the entire program, implementations are not required to avoid data races on it (**??**).

28.3.1.1 Types [locale.types]

28.3.1.1.1 Type `locale::category` [locale.category]

```
using category = int;
```

1 *Valid* `category` values include the `locale` member bitmask elements `collate`, `ctype`, `monetary`, `numeric`, `time`, and `messages`, each of which represents a single locale category. In addition, `locale` member bitmask constant `none` is defined as zero and represents no category. And `locale` member bitmask constant `all` is defined such that the expression

```
(collate | ctype | monetary | numeric | time | messages | all) == all
```

is `true`, and represents the union of all categories. Further, the expression `(X | Y)`, where `X` and `Y` each represent a single category, represents the union of the two categories.

2 `locale` member functions expecting a `category` argument require one of the `category` values defined above, or the union of two or more such values. Such a `category` value identifies a set of locale categories. Each locale category, in turn, identifies a set of locale facets, including at least those shown in Table 101.

Table 101: Locale category facets [tab:locale.category.facets]

Category	Includes facets
collate	`collate<char>`, `collate<wchar_t>`

ctype	`ctype<char>`, `ctype<wchar_t>`
	`codecvt<char, char, mbstate_t>`
	`codecvt<char16_t, char8_t, mbstate_t>`
	`codecvt<char32_t, char8_t, mbstate_t>`
	`codecvt<wchar_t, char, mbstate_t>`

monetary	`moneypunct<char>`, `moneypunct<wchar_t>`
	`moneypunct<char, true>`, `moneypunct<wchar_t, true>`
	`money_get<char>`, `money_get<wchar_t>`
	`money_put<char>`, `money_put<wchar_t>`

numeric	`numpunct<char>`, `numpunct<wchar_t>`
	`num_get<char>`, `num_get<wchar_t>`
	`num_put<char>`, `num_put<wchar_t>`

time	`time_get<char>`, `time_get<wchar_t>` `time_put<char>`, `time_put<wchar_t>`
messages	`messages<char>`, `messages<wchar_t>`

3 For any locale `loc` either constructed, or returned by `locale::classic()`, and any facet `Facet` shown in Table 101, `has_facet<Facet>(loc)` is `true`. Each `locale` member function which takes a `locale::category` argument operates on the corresponding set of facets.

4 An implementation is required to provide those specializations for facet templates identified as members of a category, and for those shown in Table 102.

5 The provided implementation of members of facets `num_get<charT>` and `num_put<charT>` calls `use_facet<F>(l)` only for facet `F` of types `numpunct<charT>` and `ctype<charT>`, and for locale `l` the value obtained by calling member `getloc()` on the `ios_base&` argument to these functions.

6 In declarations of facets, a template parameter with name `InputIterator` or `OutputIterator` indicates the set of all possible specializations on parameters that meet the *Cpp17InputIterator* requirements or *Cpp17OutputIterator* requirements, respectively (**??**). A template parameter with name `C` represents the set of types containing `char`, `wchar_t`, and any other implementation-defined character types that meet the

Table 102: Required specializations [tab:locale.spec]

Category	Includes facets
collate	`collate_byname<char>`, `collate_byname<wchar_t>`

ctype	`ctype_byname<char>`, `ctype_byname<wchar_t>`
	`codecvt_byname<char, char, mbstate_t>`
	`codecvt_byname<char16_t, char8_t, mbstate_t>`
	`codecvt_byname<char32_t, char8_t, mbstate_t>`
	`codecvt_byname<wchar_t, char, mbstate_t>`

monetary	`moneypunct_byname<char, International>`
	`moneypunct_byname<wchar_t, International>`
	`money_get<C, InputIterator>`
	`money_put<C, OutputIterator>`

numeric	`numpunct_byname<char>`, `numpunct_byname<wchar_t>` `num_get<C, InputIterator>`, `num_put<C, OutputIterator>`

time	`time_get<char, InputIterator>`
	`time_get_byname<char, InputIterator>`
	`time_get<wchar_t, InputIterator>`
	`time_get_byname<wchar_t, InputIterator>`
	`time_put<char, OutputIterator>`
	`time_put_byname<char, OutputIterator>`
	`time_put<wchar_t, OutputIterator>`
	`time_put_byname<wchar_t, OutputIterator>`

messages	`messages_byname<char>`, `messages_byname<wchar_t>`

requirements for a character on which any of the iostream components can be instantiated. A template parameter with name `International` represents the set of all possible specializations on a bool parameter.

28.3.1.1.2 Class `locale::facet` [locale.facet]

```
namespace std {
  class locale::facet {
  protected:
    explicit facet(size_t refs = 0);
    virtual ~facet();
    facet(const facet&) = delete;
    void operator=(const facet&) = delete;
  };
}
```

1 Class `facet` is the base class for locale feature sets. A class is a *facet* if it is publicly derived from another facet, or if it is a class derived from `locale::facet` and contains a publicly accessible declaration as follows:[259]

```
static ::std::locale::id id;
```

2 Template parameters in this Clause which are required to be facets are those named `Facet` in declarations. A program that passes a type that is *not* a facet, or a type that refers to a volatile-qualified facet, as an (explicit or deduced) template parameter to a locale function expecting a facet, is ill-formed. A const-qualified facet is a valid template argument to any locale function that expects a Facet template parameter.

3 The `refs` argument to the constructor is used for lifetime management. For `refs == 0`, the implementation performs `delete static_cast<locale::facet*>(f)` (where `f` is a pointer to the facet) when the last `locale` object containing the facet is destroyed; for `refs == 1`, the implementation never destroys the facet.

4 Constructors of all facets defined in this Clause take such an argument and pass it along to their `facet` base class constructor. All one-argument constructors defined in this Clause are *explicit*, preventing their participation in automatic conversions.

5 For some standard facets a standard "`..._byname`" class, derived from it, implements the virtual function semantics equivalent to that facet of the locale constructed by `locale(const char*)` with the same name.

259) This is a complete list of requirements; there are no other requirements. Thus, a facet class need not have a public copy constructor, assignment, default constructor, destructor, etc.

Each such facet provides a constructor that takes a `const char*` argument, which names the locale, and a `refs` argument, which is passed to the base class constructor. Each such facet also provides a constructor that takes a `string` argument `str` and a `refs` argument, which has the same effect as calling the first constructor with the two arguments `str.c_str()` and `refs`. If there is no "`..._byname`" version of a facet, the base class implements named locale semantics itself by reference to other facets.

28.3.1.1.3 Class `locale::id` [locale.id]

```
namespace std {
  class locale::id {
  public:
    id();
    void operator=(const id&) = delete;
    id(const id&) = delete;
  };
}
```

1 The class `locale::id` provides identification of a locale facet interface, used as an index for lookup and to encapsulate initialization.

2 [*Note*: Because facets are used by iostreams, potentially while static constructors are running, their initialization cannot depend on programmed static initialization. One initialization strategy is for `locale` to initialize each facet's `id` member the first time an instance of the facet is installed into a locale. This depends only on static storage being zero before constructors run (**??**). — *end note*]

28.3.1.2 Constructors and destructor [locale.cons]

`locale() noexcept;`

1 Default constructor: a snapshot of the current global locale.

2 *Effects:* Constructs a copy of the argument last passed to `locale::global(locale&)`, if it has been called; else, the resulting facets have virtual function semantics identical to those of `locale::classic()`. [*Note*: This constructor yields a copy of the current global locale. It is commonly used as the default value for arguments of functions that take a `const locale&` argument. — *end note*]

`locale(const locale& other) noexcept;`

3 *Effects:* Constructs a locale which is a copy of `other`.

`explicit locale(const char* std_name);`

4 *Effects:* Constructs a locale using standard C locale names, e.g., `"POSIX"`. The resulting locale implements semantics defined to be associated with that name.

5 *Throws:* `runtime_error` if the argument is not valid, or is null.

6 *Remarks:* The set of valid string argument values is `"C"`, `""`, and any implementation-defined values.

`explicit locale(const string& std_name);`

7 *Effects:* The same as `locale(std_name.c_str())`.

`locale(const locale& other, const char* std_name, category);`

8 *Effects:* Constructs a locale as a copy of `other` except for the facets identified by the `category` argument, which instead implement the same semantics as `locale(std_name)`.

9 *Throws:* `runtime_error` if the argument is not valid, or is null.

10 *Remarks:* The locale has a name if and only if `other` has a name.

`locale(const locale& other, const string& std_name, category cat);`

11 *Effects:* The same as `locale(other, std_name.c_str(), cat)`.

`template<class Facet> locale(const locale& other, Facet* f);`

12 *Effects:* Constructs a locale incorporating all facets from the first argument except that of type `Facet`, and installs the second argument as the remaining facet. If `f` is null, the resulting object is a copy of `other`.

13 *Remarks:* The resulting locale has no name.

```
locale(const locale& other, const locale& one, category cats);
```

14 *Effects:* Constructs a locale incorporating all facets from the first argument except those that implement `cats`, which are instead incorporated from the second argument.

15 *Remarks:* The resulting locale has a name if and only if the first two arguments have names.

```
const locale& operator=(const locale& other) noexcept;
```

16 *Effects:* Creates a copy of `other`, replacing the current value.

17 *Returns:* `*this`.

```
~locale();
```

18 A non-virtual destructor that throws no exceptions.

28.3.1.3 Members [locale.members]

```
template<class Facet> locale combine(const locale& other) const;
```

1 *Effects:* Constructs a locale incorporating all facets from `*this` except for that one facet of `other` that is identified by `Facet`.

2 *Returns:* The newly created locale.

3 *Throws:* `runtime_error` if `has_facet<Facet>(other)` is `false`.

4 *Remarks:* The resulting locale has no name.

```
basic_string<char>string name() const;
```

5 *Returns:* The name of `*this`, if it has one; otherwise, the string `"*"`.

28.3.1.4 Operators [locale.operators]

```
bool operator==(const locale& other) const;
```

1 *Returns:* `true` if both arguments are the same locale, or one is a copy of the other, or each has a name and the names are identical; `false` otherwise.

```
template<class charT, class traits, class Allocator>
  bool operator()(const basic_string<charT, traits, Allocator>& s1,
                  const basic_string<charT, traits, Allocator>& s2) const;
```

2 *Effects:* Compares two strings according to the `collate<charT>` facet.

3 *Remarks:* This member operator template (and therefore `locale` itself) meets the requirements for a comparator predicate template argument (**??**) applied to strings.

4 *Returns:*

```
use_facet<collate<charT>>(*this).compare(s1.data(), s1.data() + s1.size(),
                                         s2.data(), s2.data() + s2.size()) < 0
```

5 [*Example*: A vector of strings `v` can be collated according to collation rules in locale `loc` simply by (**??**, **??**):

```
std::sort(v.begin(), v.end(), loc);
```

— *end example*]

28.3.1.5 Static members [locale.statics]

```
static locale global(const locale& loc);
```

[Editor's note: The old P1 gets an *Effects:* , and the old P2 gets merged into that. P3 split off from the old P2]

1 *Effects:* Sets the global locale to its argument.

2 *Effects:* Causes future calls to the constructor `locale()` to return a copy of the argument. If the argument has a name, does

```
setlocale(LC_ALL, loc.name().c_str());
```

otherwise, the effect on the C locale, if any, is implementation-defined. No library function other than `locale::global()` shall affect the value returned by `locale()`.

3 *Remarks:* No library function other than `locale::global()` affects the value returned by `locale()`.

[*Note*: See 28.5 for data race considerations when `setlocale` is invoked. — *end note*]

4 *Returns:* The previous value of `locale()`.

```
static const locale& classic();
```

5 The `"C"` locale.

6 *Returns:* A locale that implements the classic `"C"` locale semantics, equivalent to the value `locale("C")`.

7 *Remarks:* This locale, its facets, and their member functions, do not change with time.

28.3.2 `locale` globals [locale.global.templates]

```
template<class Facet> const Facet& use_facet(const locale& loc);
```

1 ~~*Requires:*~~ *Mandates:* `Facet` is a facet class whose definition contains the public static member `id` as defined in 28.3.1.1.2.

2 *Returns:* A reference to the corresponding facet of `loc`, if present.

3 *Throws:* `bad_cast` if `has_facet<Facet>(loc)` is `false`.

4 *Remarks:* The reference returned remains valid at least as long as any copy of `loc` exists.

```
template<class Facet> bool has_facet(const locale& loc) noexcept;
```

5 *Returns:* `true` if the facet requested is present in `loc`; otherwise `false`.

28.3.3 Convenience interfaces [locale.convenience]

28.3.3.1 Character classification [classification]

```
template<class charT> bool isspace (charT c, const locale& loc);
template<class charT> bool isprint (charT c, const locale& loc);
template<class charT> bool iscntrl (charT c, const locale& loc);
template<class charT> bool isupper (charT c, const locale& loc);
template<class charT> bool islower (charT c, const locale& loc);
template<class charT> bool isalpha (charT c, const locale& loc);
template<class charT> bool isdigit (charT c, const locale& loc);
template<class charT> bool ispunct (charT c, const locale& loc);
template<class charT> bool isxdigit(charT c, const locale& loc);
template<class charT> bool isalnum (charT c, const locale& loc);
template<class charT> bool isgraph (charT c, const locale& loc);
template<class charT> bool isblank (charT c, const locale& loc);
```

1 Each of these functions `is`*F* returns the result of the expression:

```
use_facet<ctype<charT>>(loc).is(ctype_base::F, c)
```

where *F* is the `ctype_base::mask` value corresponding to that function (28.4.1).[260]

28.3.3.2 Conversions [conversions]

28.3.3.2.1 Character conversions [conversions.character]

```
template<class charT> charT toupper(charT c, const locale& loc);
```

1 *Returns:* `use_facet<ctype<charT>>(loc).toupper(c)`.

```
template<class charT> charT tolower(charT c, const locale& loc);
```

2 *Returns:* `use_facet<ctype<charT>>(loc).tolower(c)`.

28.4 Standard `locale` categories [locale.categories]

1 Each of the standard categories includes a family of facets. Some of these implement formatting or parsing of a datum, for use by standard or users' iostream operators `<<` and `>>`, as members `put()` and `get()`, respectively.

260) When used in a loop, it is faster to cache the `ctype<>` facet and use it directly, or use the vector form of `ctype<>::is`.

Each such member function takes an `ios_base&` argument whose members `flags()`, `precision()`, and `width()`, specify the format of the corresponding datum (**??**). Those functions which need to use other facets call its member `getloc()` to retrieve the locale imbued there. Formatting facets use the character argument `fill` to fill out the specified width where necessary.

2 The `put()` members make no provision for error reporting. (Any failures of the OutputIterator argument can be extracted from the returned iterator.) The `get()` members take an `ios_base::iostate&` argument whose value they ignore, but set to `ios_base::failbit` in case of a parse error.

3 Within this clause it is unspecified whether one virtual function calls another virtual function.

28.4.1 The ctype category [category.ctype]

```
namespace std {
  class ctype_base {
  public:
    using mask = see below;

    // numeric values are for exposition only.
    static const mask space  = 1 << 0;
    static const mask print  = 1 << 1;
    static const mask cntrl  = 1 << 2;
    static const mask upper  = 1 << 3;
    static const mask lower  = 1 << 4;
    static const mask alpha  = 1 << 5;
    static const mask digit  = 1 << 6;
    static const mask punct  = 1 << 7;
    static const mask xdigit = 1 << 8;
    static const mask blank  = 1 << 9;
    static const mask alnum  = alpha | digit;
    static const mask graph  = alnum | punct;
  };
}
```

1 The type `mask` is a bitmask type (**??**).

28.4.1.1 Class template ctype [locale.ctype]

```
namespace std {
  template<class charT>
    class ctype : public locale::facet, public ctype_base {
    public:
      using char_type = charT;

      explicit ctype(size_t refs = 0);

      bool         is(mask m, charT c) const;
      const charT* is(const charT* low, const charT* high, mask* vec) const;
      const charT* scan_is(mask m, const charT* low, const charT* high) const;
      const charT* scan_not(mask m, const charT* low, const charT* high) const;
      charT        toupper(charT c) const;
      const charT* toupper(charT* low, const charT* high) const;
      charT        tolower(charT c) const;
      const charT* tolower(charT* low, const charT* high) const;

      charT        widen(char c) const;
      const char*  widen(const char* low, const char* high, charT* to) const;
      char         narrow(charT c, char dfault) const;
      const charT* narrow(const charT* low, const charT* high, char dfault, char* to) const;

      static locale::id id;

    protected:
      ~ctype();
      virtual bool         do_is(mask m, charT c) const;
      virtual const charT* do_is(const charT* low, const charT* high, mask* vec) const;
```

```
    virtual const charT* do_scan_is(mask m, const charT* low, const charT* high) const;
    virtual const charT* do_scan_not(mask m, const charT* low, const charT* high) const;
    virtual charT        do_toupper(charT) const;
    virtual const charT* do_toupper(charT* low, const charT* high) const;
    virtual charT        do_tolower(charT) const;
    virtual const charT* do_tolower(charT* low, const charT* high) const;
    virtual charT        do_widen(char) const;
    virtual const char*  do_widen(const char* low, const char* high, charT* dest) const;
    virtual char         do_narrow(charT, char dfault) const;
    virtual const charT* do_narrow(const charT* low, const charT* high,
                                   char dfault, char* dest) const;
  };
}
```

1 Class `ctype` encapsulates the C library `<cctype>` features. `istream` members are required to use `ctype<>` for character classing during input parsing.

2 The specializations required in Table 101 (28.3.1.1.1), namely `ctype<char>` and `ctype<wchar_t>`, implement character classing appropriate to the implementation's native character set.

28.4.1.1.1 `ctype` members [locale.ctype.members]

```
bool         is(mask m, charT c) const;
const charT* is(const charT* low, const charT* high, mask* vec) const;
```

1 *Returns:* `do_is(m, c)` or `do_is(low, high, vec)`.

```
const charT* scan_is(mask m, const charT* low, const charT* high) const;
```

2 *Returns:* `do_scan_is(m, low, high)`.

```
const charT* scan_not(mask m, const charT* low, const charT* high) const;
```

3 *Returns:* `do_scan_not(m, low, high)`.

```
charT        toupper(charT) const;
const charT* toupper(charT* low, const charT* high) const;
```

4 *Returns:* `do_toupper(c)` or `do_toupper(low, high)`.

```
charT        tolower(charT c) const;
const charT* tolower(charT* low, const charT* high) const;
```

5 *Returns:* `do_tolower(c)` or `do_tolower(low, high)`.

```
charT       widen(char c) const;
const char* widen(const char* low, const char* high, charT* to) const;
```

6 *Returns:* `do_widen(c)` or `do_widen(low, high, to)`.

```
char         narrow(charT c, char dfault) const;
const charT* narrow(const charT* low, const charT* high, char dfault, char* to) const;
```

7 *Returns:* `do_narrow(c, dfault)` or `do_narrow(low, high, dfault, to)`.

28.4.1.1.2 `ctype` virtual functions [locale.ctype.virtuals]

```
bool         do_is(mask m, charT c) const;
const charT* do_is(const charT* low, const charT* high, mask* vec) const;
```

1 *Effects:* Classifies a character or sequence of characters. For each argument character, identifies a value `M` of type `ctype_base::mask`. The second form identifies a value `M` of type `ctype_base::mask` for each `*p` where `(low <= p && p < high)`, and places it into `vec[p - low]`.

2 *Returns:* The first form returns the result of the expression `(M & m) != 0`; i.e., `true` if the character has the characteristics specified. The second form returns `high`.

```
const charT* do_scan_is(mask m, const charT* low, const charT* high) const;
```

3 *Effects:* Locates a character in a buffer that conforms to a classification `m`.

4 *Returns:* The smallest pointer `p` in the range `[low, high)` such that `is(m, *p)` would return `true`; otherwise, returns `high`.

```
const charT* do_scan_not(mask m, const charT* low, const charT* high) const;
```

5 *Effects:* Locates a character in a buffer that fails to conform to a classification `m`.

6 *Returns:* The smallest pointer `p`, if any, in the range `[low, high)` such that `is(m, *p)` would return `false`; otherwise, returns `high`.

```
charT        do_toupper(charT c) const;
const charT* do_toupper(charT* low, const charT* high) const;
```

7 *Effects:* Converts a character or characters to upper case. The second form replaces each character `*p` in the range `[low, high)` for which a corresponding upper-case character exists, with that character.

8 *Returns:* The first form returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form returns `high`.

```
charT        do_tolower(charT c) const;
const charT* do_tolower(charT* low, const charT* high) const;
```

9 *Effects:* Converts a character or characters to lower case. The second form replaces each character `*p` in the range `[low, high)` and for which a corresponding lower-case character exists, with that character.

10 *Returns:* The first form returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form returns `high`.

```
charT        do_widen(char c) const;
const char*  do_widen(const char* low, const char* high, charT* dest) const;
```

11 *Effects:* Applies the simplest reasonable transformation from a `char` value or sequence of `char` values to the corresponding `charT` value or values.[261] The only characters for which unique transformations are required are those in the basic source character set (**??**).

For any named `ctype` category with a `ctype <charT>` facet `ctc` and valid `ctype_base::mask` value `M`, `(ctc.is(M, c) || !is(M, do_widen(c)) )` is `true`.[262]

The second form transforms each character `*p` in the range `[low, high)`, placing the result in `dest[p - low]`.

12 *Returns:* The first form returns the transformed value. The second form returns `high`.

```
char         do_narrow(charT c, char dfault) const;
const charT* do_narrow(const charT* low, const charT* high, char dfault, char* dest) const;
```

13 *Effects:* Applies the simplest reasonable transformation from a `charT` value or sequence of `charT` values to the corresponding `char` value or values.

For any character `c` in the basic source character set (**??**) the transformation is such that

```
do_widen(do_narrow(c, 0)) == c
```

For any named `ctype` category with a `ctype<char>` facet `ctc` however, and `ctype_base::mask` value `M`,

```
(is(M, c) || !ctc.is(M, do_narrow(c, dfault)) )
```

is `true` (unless `do_narrow` returns `dfault`). In addition, for any digit character `c`, the expression `(do_narrow(c, dfault) - '0')` evaluates to the digit value of the character. The second form transforms each character `*p` in the range `[low, high)`, placing the result (or `dfault` if no simple transformation is readily available) in `dest[p - low]`.

14 *Returns:* The first form returns the transformed value; or `dfault` if no mapping is readily available. The second form returns `high`.

261) The char argument of `do_widen` is intended to accept values derived from character literals for conversion to the locale's encoding.

262) In other words, the transformed character is not a member of any character classification that `c` is not also a member of.

28.4.1.2 Class template ctype_byname [locale.ctype.byname]

```
namespace std {
  template<class charT>
    class ctype_byname : public ctype<charT> {
    public:
      using mask = typename ctype<charT>::mask;
      explicit ctype_byname(const char*, size_t refs = 0);
      explicit ctype_byname(const string&, size_t refs = 0);

    protected:
      ~ctype_byname();
    };
}
```

28.4.1.3 ctype<char> specialization [facet.ctype.special]

```
namespace std {
  template<>
    class ctype<char> : public locale::facet, public ctype_base {
    public:
      using char_type = char;

      explicit ctype(const mask* tab = nullptr, bool del = false, size_t refs = 0);

      bool is(mask m, char c) const;
      const char* is(const char* low, const char* high, mask* vec) const;
      const char* scan_is (mask m, const char* low, const char* high) const;
      const char* scan_not(mask m, const char* low, const char* high) const;

      char        toupper(char c) const;
      const char* toupper(char* low, const char* high) const;
      char        tolower(char c) const;
      const char* tolower(char* low, const char* high) const;

      char  widen(char c) const;
      const char* widen(const char* low, const char* high, char* to) const;
      char  narrow(char c, char dfault) const;
      const char* narrow(const char* low, const char* high, char dfault, char* to) const;

      static locale::id id;
      static const size_t table_size = implementation-defined;

      const mask* table() const noexcept;
      static const mask* classic_table() noexcept;

    protected:
      ~ctype();
      virtual char        do_toupper(char c) const;
      virtual const char* do_toupper(char* low, const char* high) const;
      virtual char        do_tolower(char c) const;
      virtual const char* do_tolower(char* low, const char* high) const;

      virtual char        do_widen(char c) const;
      virtual const char* do_widen(const char* low, const char* high, char* to) const;
      virtual char        do_narrow(char c, char dfault) const;
      virtual const char* do_narrow(const char* low, const char* high,
                                    char dfault, char* to) const;
    };
}
```

1 A specialization `ctype<char>` is provided so that the member functions on type `char` can be implemented inline.[263] The implementation-defined value of member `table_size` is at least 256.

263) Only the `char` (not `unsigned char` and `signed char`) form is provided. The specialization is specified in the standard, and not left as an implementation detail, because it affects the derivation interface for `ctype<char>`.

28.4.1.3.1 Destructor [facet.ctype.char.dtor]

```
~ctype();
```

1 *Effects:* If the constructor's first argument was nonzero, and its second argument was `true`, does `delete [] table()`.

28.4.1.3.2 Members [facet.ctype.char.members]

1 In the following member descriptions, for `unsigned char` values `v` where `v >= table_size`, `table()[v]` is assumed to have an implementation-specific value (possibly different for each such value `v`) without performing the array lookup.

```
explicit ctype(const mask* tbl = nullptr, bool del = false, size_t refs = 0);
```

2 *Requires:* `tbl` either 0 or an array of at least `table_size` elements.

3 *Expects:* Either `tbl == nullptr` is true or `[tbl, tbl+table_size)` is a valid range.

4 *Effects:* Passes its `refs` argument to its base class constructor.

```
bool        is(mask m, char c) const;
const char* is(const char* low, const char* high, mask* vec) const;
```

5 *Effects:* The second form, for all `*p` in the range `[low, high)`, assigns into `vec[p - low]` the value `table()[(unsigned char)*p]`.

6 *Returns:* The first form returns `table()[(unsigned char)c] & m`; the second form returns `high`.

```
const char* scan_is(mask m, const char* low, const char* high) const;
```

7 *Returns:* The smallest `p` in the range `[low, high)` such that

```
table()[(unsigned char) *p] & m
```

is `true`.

```
const char* scan_not(mask m, const char* low, const char* high) const;
```

8 *Returns:* The smallest `p` in the range `[low, high)` such that

```
table()[(unsigned char) *p] & m
```

is `false`.

```
char        toupper(char c) const;
const char* toupper(char* low, const char* high) const;
```

9 *Returns:* `do_toupper(c)` or `do_toupper(low, high)`, respectively.

```
char        tolower(char c) const;
const char* tolower(char* low, const char* high) const;
```

10 *Returns:* `do_tolower(c)` or `do_tolower(low, high)`, respectively.

```
char  widen(char c) const;
const char* widen(const char* low, const char* high, char* to) const;
```

11 *Returns:* `do_widen(c)` or `do_widen(low, high, to)`, respectively.

```
char        narrow(char c, char dfault) const;
const char* narrow(const char* low, const char* high, char dfault, char* to) const;
```

12 *Returns:* `do_narrow(c, dfault)` or `do_narrow(low, high, dfault, to)`, respectively.

```
const mask* table() const noexcept;
```

13 *Returns:* The first constructor argument, if it was nonzero, otherwise `classic_table()`.

28.4.1.3.3 Static members [facet.ctype.char.statics]

```
static const mask* classic_table() noexcept;
```

1 *Returns:* A pointer to the initial element of an array of size `table_size` which represents the classifications of characters in the `"C"` locale.

28.4.1.3.4 Virtual functions [facet.ctype.char.virtuals]

```
char        do_toupper(char) const;
const char* do_toupper(char* low, const char* high) const;
char        do_tolower(char) const;
const char* do_tolower(char* low, const char* high) const;

virtual char        do_widen(char c) const;
virtual const char* do_widen(const char* low, const char* high, char* to) const;
virtual char        do_narrow(char c, char dfault) const;
virtual const char* do_narrow(const char* low, const char* high,
                              char dfault, char* to) const;
```

[1] These functions are described identically as those members of the same name in the `ctype` class template (28.4.1.1.1).

28.4.1.4 Class template `codecvt` [locale.codecvt]

```
namespace std {
  class codecvt_base {
  public:
    enum result { ok, partial, error, noconv };
  };

  template<class internT, class externT, class stateT>
    class codecvt : public locale::facet, public codecvt_base {
    public:
      using intern_type = internT;
      using extern_type = externT;
      using state_type  = stateT;

      explicit codecvt(size_t refs = 0);

      result out(
          stateT& state,
          const internT* from, const internT* from_end, const internT*& from_next,
                externT*   to,       externT*   to_end,       externT*&   to_next) const;

      result unshift(
          stateT& state,
                externT*    to,      externT*   to_end,       externT*&   to_next) const;

      result in(
          stateT& state,
          const externT* from, const externT* from_end, const externT*& from_next,
                internT*   to,       internT*   to_end,       internT*&   to_next) const;

      int encoding() const noexcept;
      bool always_noconv() const noexcept;
      int length(stateT&, const externT* from, const externT* end, size_t max) const;
      int max_length() const noexcept;

      static locale::id id;

    protected:
      ~codecvt();
      virtual result do_out(
          stateT& state,
          const internT* from, const internT* from_end, const internT*& from_next,
                externT* to,         externT*   to_end,       externT*&   to_next) const;
      virtual result do_in(
          stateT& state,
          const externT* from, const externT* from_end, const externT*& from_next,
                internT* to,         internT*   to_end,       internT*&   to_next) const;
```

```
    virtual result do_unshift(
        stateT& state,
              externT* to,         externT*   to_end,       externT*&   to_next) const;

    virtual int do_encoding() const noexcept;
    virtual bool do_always_noconv() const noexcept;
    virtual int do_length(stateT&, const externT* from, const externT* end, size_t max) const;
    virtual int do_max_length() const noexcept;
  };
}
```

1 The class `codecvt<internT, externT, stateT>` is for use when converting from one character encoding to another, such as from wide characters to multibyte characters or between wide character encodings such as Unicode and EUC.

2 The `stateT` argument selects the pair of character encodings being mapped between.

3 The specializations required in Table 101 (28.3.1.1.1) convert the implementation-defined native character set. `codecvt<char, char, mbstate_t>` implements a degenerate conversion; it does not convert at all. The specialization `codecvt<char16_t, char8_t, mbstate_t>` converts between the UTF-16 and UTF-8 encoding forms, and the specialization `codecvt <char32_t, char8_t, mbstate_t>` converts between the UTF-32 and UTF-8 encoding forms. `codecvt<wchar_t, char, mbstate_t>` converts between the native character sets for ordinary and wide characters. Specializations on `mbstate_t` perform conversion between encodings known to the library implementer. Other encodings can be converted by specializing on a program-defined `stateT` type. Objects of type `stateT` can contain any state that is useful to communicate to or from the specialized `do_in` or `do_out` members.

28.4.1.4.1 Members [locale.codecvt.members]

```
result out(
    stateT& state,
    const internT* from, const internT* from_end, const internT*& from_next,
    externT* to, externT* to_end, externT*& to_next) const;
```

1 *Returns:* `do_out(state, from, from_end, from_next, to, to_end, to_next)`.

```
result unshift(stateT& state, externT* to, externT* to_end, externT*& to_next) const;
```

2 *Returns:* `do_unshift(state, to, to_end, to_next)`.

```
result in(
    stateT& state,
    const externT* from, const externT* from_end, const externT*& from_next,
    internT* to, internT* to_end, internT*& to_next) const;
```

3 *Returns:* `do_in(state, from, from_end, from_next, to, to_end, to_next)`.

```
int encoding() const noexcept;
```

4 *Returns:* `do_encoding()`.

```
bool always_noconv() const noexcept;
```

5 *Returns:* `do_always_noconv()`.

```
int length(stateT& state, const externT* from, const externT* from_end, size_t max) const;
```

6 *Returns:* `do_length(state, from, from_end, max)`.

```
int max_length() const noexcept;
```

7 *Returns:* `do_max_length()`.

28.4.1.4.2 Virtual functions [locale.codecvt.virtuals]

```
result do_out(
    stateT& state,
    const internT* from, const internT* from_end, const internT*& from_next,
    externT* to, externT* to_end, externT*& to_next) const;
```

```
result do_in(
    stateT& state,
    const externT* from, const externT* from_end, const externT*& from_next,
    internT* to, internT* to_end, internT*& to_next) const;
```

1 ~~*Requires:*~~ *Expects:* `(from <= from_end && to <= to_end)` is well-defined and `true`; `state` is initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

2 *Effects:* Translates characters in the source range `[from, from_end)`, placing the results in sequential positions starting at destination `to`. Converts no more than `(from_end - from)` source elements, and stores no more than `(to_end - to)` destination elements.

Stops if it encounters a character it cannot convert. It always leaves the `from_next` and `to_next` pointers pointing one beyond the last element successfully converted. If returns `noconv`, `internT` and `externT` are the same type and the converted sequence is identical to the input sequence `[from, from_next)`. `to_next` is set equal to `to`, the value of `state` is unchanged, and there are no changes to the values in `[to, to_end)`.

3 A `codecvt` facet that is used by `basic_filebuf` (??) shall have the property that if

```
do_out(state, from, from_end, from_next, to, to_end, to_next)
```

would return `ok`, where `from != from_end`, then

```
do_out(state, from, from + 1, from_next, to, to_end, to_next)
```

shall also return `ok`, and that if

```
do_in(state, from, from_end, from_next, to, to_end, to_next)
```

would return `ok`, where `to != to_end`, then

```
do_in(state, from, from_end, from_next, to, to + 1, to_next)
```

shall also return `ok`.[264] [*Note*: As a result of operations on `state`, it can return `ok` or `partial` and set `from_next == from` and `to_next != to`. — *end note*]

4 *Remarks:* Its operations on `state` are unspecified. [*Note*: This argument can be used, for example, to maintain shift state, to specify conversion options (such as count only), or to identify a cache of seek offsets. — *end note*]

5 *Returns:* An enumeration value, as summarized in Table 103.

Table 103: `do_in/do_out` result values [tab:locale.codecvt.inout]

Value	Meaning
`ok`	completed the conversion
`partial`	not all source characters converted
`error`	encountered a character in `[from, from_end)` that it could not convert
`noconv`	`internT` and `externT` are the same type, and input sequence is identical to converted sequence

A return value of `partial`, if `(from_next == from_end)`, indicates that either the destination sequence has not absorbed all the available destination elements, or that additional source elements are needed before another destination element can be produced.

```
result do_unshift(stateT& state, externT* to, externT* to_end, externT*& to_next) const;
```

6 ~~*Requires:*~~ *Expects:* `(to <= to_end)` is well-defined and `true`; state is initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

7 *Effects:* Places characters starting at `to` that should be appended to terminate a sequence when the current `stateT` is given by `state`.[265] Stores no more than `(to_end - to)` destination elements, and leaves the `to_next` pointer pointing one beyond the last element successfully stored.

264) Informally, this means that `basic_filebuf` assumes that the mappings from internal to external characters is 1 to N: a `codecvt` facet that is used by `basic_filebuf` must be able to translate characters one internal character at a time.
265) Typically these will be characters to return the state to `stateT()`.

8 *Returns:* An enumeration value, as summarized in Table 104.

Table 104: `do_unshift` result values [tab:locale.codecvt.unshift]

Value	Meaning
`ok`	completed the sequence
`partial`	space for more than `to_end - to` destination elements was needed to terminate a sequence given the value of `state`
`error`	an unspecified error has occurred
`noconv`	no termination is needed for this `state_type`

```
int do_encoding() const noexcept;
```

9 *Returns:* `-1` if the encoding of the `externT` sequence is state-dependent; else the constant number of `externT` characters needed to produce an internal character; or `0` if this number is not a constant.[266]

```
bool do_always_noconv() const noexcept;
```

10 *Returns:* `true` if `do_in()` and `do_out()` return `noconv` for all valid argument values. `codecvt<char, char, mbstate_t>` returns `true`.

```
int do_length(stateT& state, const externT* from, const externT* from_end, size_t max) const;
```

11 ~~*Requires:*~~ *Expects:* `(from <= from_end)` is well-defined and `true`; `state` is initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

12 *Effects:* The effect on the `state` argument is "as if" it called `do_in(state, from, from_end, from, to, to+max, to)` for `to` pointing to a buffer of at least `max` elements.

13 *Returns:* `(from_next-from)` where `from_next` is the largest value in the range `[from, from_end]` such that the sequence of values in the range `[from, from_next)` represents `max` or fewer valid complete characters of type `internT`. The specialization `codecvt<char, char, mbstate_t>`, returns the lesser of `max` and `(from_end-from)`.

```
int do_max_length() const noexcept;
```

14 *Returns:* The maximum value that `do_length(state, from, from_end, 1)` can return for any valid range `[from, from_end)` and `stateT` value `state`. The specialization `codecvt<char, char, mbstate_t>::do_max_length()` returns 1.

28.4.1.5 Class template `codecvt_byname` [locale.codecvt.byname]

```
namespace std {
  template<class internT, class externT, class stateT>
    class codecvt_byname : public codecvt<internT, externT, stateT> {
    public:
      explicit codecvt_byname(const char*, size_t refs = 0);
      explicit codecvt_byname(const string&, size_t refs = 0);

    protected:
      ~codecvt_byname();
    };
}
```

28.4.2 The numeric category [category.numeric]

1 The classes `num_get<>` and `num_put<>` handle numeric formatting and parsing. Virtual functions are provided for several numeric types. Implementations may (but are not required to) delegate extraction of smaller types to extractors for larger types.[267]

266) If `encoding()` yields `-1`, then more than `max_length()` `externT` elements may be consumed when producing a single `internT` character, and additional `externT` elements may appear at the end of a sequence after those that yield the final `internT` character.

267) Parsing `"-1"` correctly into, e.g., an `unsigned short` requires that the corresponding member `get()` at least extract the sign before delegating.

[2] All specifications of member functions for `num_put` and `num_get` in the subclauses of 28.4.2 only apply to the specializations required in Tables 101 and 102 (28.3.1.1.1), namely `num_get<char>`, `num_get<wchar_t>`, `num_get<C, InputIterator>`, `num_put<char>`, `num_put<wchar_t>`, and `num_put<C, OutputIterator>`. These specializations refer to the `ios_base&` argument for formatting specifications (28.4), and to its imbued locale for the `numpunct<>` facet to identify all numeric punctuation preferences, and also for the `ctype<>` facet to perform character classification.

[3] Extractor and inserter members of the standard iostreams use `num_get<>` and `num_put<>` member functions for formatting and parsing numeric values (**??**, **??**).

28.4.2.1 Class template `num_get` [locale.num.get]

```
namespace std {
  template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class num_get : public locale::facet {
    public:
      using char_type = charT;
      using iter_type = InputIterator;

      explicit num_get(size_t refs = 0);

      iter_type get(iter_type in, iter_type end, ios_base&,
                    ios_base::iostate& err, bool& v) const;
      iter_type get(iter_type in, iter_type end, ios_base&,
                    ios_base::iostate& err, long& v) const;
      iter_type get(iter_type in, iter_type end, ios_base&,
                    ios_base::iostate& err, long long& v) const;
      iter_type get(iter_type in, iter_type end, ios_base&,
                    ios_base::iostate& err, unsigned short& v) const;
      iter_type get(iter_type in, iter_type end, ios_base&,
                    ios_base::iostate& err, unsigned int& v) const;
      iter_type get(iter_type in, iter_type end, ios_base&,
                    ios_base::iostate& err, unsigned long& v) const;
      iter_type get(iter_type in, iter_type end, ios_base&,
                    ios_base::iostate& err, unsigned long long& v) const;
      iter_type get(iter_type in, iter_type end, ios_base&,
                    ios_base::iostate& err, float& v) const;
      iter_type get(iter_type in, iter_type end, ios_base&,
                    ios_base::iostate& err, double& v) const;
      iter_type get(iter_type in, iter_type end, ios_base&,
                    ios_base::iostate& err, long double& v) const;
      iter_type get(iter_type in, iter_type end, ios_base&,
                    ios_base::iostate& err, void*& v) const;

      static locale::id id;

    protected:
      ~num_get();
      virtual iter_type do_get(iter_type, iter_type, ios_base&,
                               ios_base::iostate& err, bool& v) const;
      virtual iter_type do_get(iter_type, iter_type, ios_base&,
                               ios_base::iostate& err, long& v) const;
      virtual iter_type do_get(iter_type, iter_type, ios_base&,
                               ios_base::iostate& err, long long& v) const;
      virtual iter_type do_get(iter_type, iter_type, ios_base&,
                               ios_base::iostate& err, unsigned short& v) const;
      virtual iter_type do_get(iter_type, iter_type, ios_base&,
                               ios_base::iostate& err, unsigned int& v) const;
      virtual iter_type do_get(iter_type, iter_type, ios_base&,
                               ios_base::iostate& err, unsigned long& v) const;
      virtual iter_type do_get(iter_type, iter_type, ios_base&,
                               ios_base::iostate& err, unsigned long long& v) const;
      virtual iter_type do_get(iter_type, iter_type, ios_base&,
                               ios_base::iostate& err, float& v) const;
```

```
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, double& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, long double& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, void*& v) const;
  };
}
```

1 The facet `num_get` is used to parse numeric values from an input sequence such as an istream.

28.4.2.1.1 Members [facet.num.get.members]

```
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, bool& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, unsigned short& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, unsigned int& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, unsigned long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, unsigned long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, float& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, long double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
              ios_base::iostate& err, void*& val) const;
```

1 *Returns:* `do_get(in, end, str, err, val)`.

28.4.2.1.2 Virtual functions [facet.num.get.virtuals]

```
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, long long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, unsigned short& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, unsigned int& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, unsigned long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, unsigned long long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, float& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, double& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, long double& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, void*& val) const;
```

1 *Effects:* Reads characters from `in`, interpreting them according to `str.flags()`, `use_facet<ctype<charT>>(loc)`, and `use_facet<numpunct<charT>>(loc)`, where `loc` is `str.getloc()`.

2 The details of this operation occur in three stages

(2.1) — Stage 1: Determine a conversion specifier

(2.2) — Stage 2: Extract characters from `in` and determine a corresponding `char` value for the format expected by the conversion specification determined in stage 1.

(2.3) — Stage 3: Store results

3 The details of the stages are presented below.

Stage 1: The function initializes local variables via

```
fmtflags flags = str.flags();
fmtflags basefield = (flags & ios_base::basefield);
fmtflags uppercase = (flags & ios_base::uppercase);
fmtflags boolalpha = (flags & ios_base::boolalpha);
```

For conversion to an integral type, the function determines the integral conversion specifier as indicated in Table 105. The table is ordered. That is, the first line whose condition is true applies.

Table 105: Integer conversions [tab:facet.num.get.int]

State	`stdio` equivalent
`basefield == oct`	`%o`
`basefield == hex`	`%X`
`basefield == 0`	`%i`
`signed` integral type	`%d`
`unsigned` integral type	`%u`

For conversions to a floating-point type the specifier is `%g`.

For conversions to `void*` the specifier is `%p`.

A length modifier is added to the conversion specification, if needed, as indicated in Table 106.

Table 106: Length modifier [tab:facet.num.get.length]

Type	Length modifier
`short`	`h`
`unsigned short`	`h`
`long`	`l`
`unsigned long`	`l`
`long long`	`ll`
`unsigned long long`	`ll`
`double`	`l`
`long double`	`L`

Stage 2: If `in == end` then stage 2 terminates. Otherwise a `charT` is taken from `in` and local variables are initialized as if by

```
char_type ct = *in;
char c = src[find(atoms, atoms + sizeof(src) - 1, ct) - atoms];
if (ct == use_facet<numpunct<charT>>(loc).decimal_point())
c = '.';
bool discard =
  ct == use_facet<numpunct<charT>>(loc).thousands_sep()
  && use_facet<numpunct<charT>>(loc).grouping().length() != 0;
```

where the values `src` and `atoms` are defined as if by:

```
static const char src[] = "0123456789abcdefxABCDEFX+-";
char_type atoms[sizeof(src)];
use_facet<ctype<charT>>(loc).widen(src, src + sizeof(src), atoms);
```

for this value of `loc`.

If `discard` is `true`, then if `'.'` has not yet been accumulated, then the position of the character is remembered, but the character is otherwise ignored. Otherwise, if `'.'` has already been accumulated, the character is discarded and Stage 2 terminates. If it is not discarded, then a check is made to determine if `c` is allowed as the next character of an input field of the conversion specifier returned by Stage 1. If so, it is accumulated.

If the character is either discarded or accumulated then `in` is advanced by `++in` and processing returns to the beginning of stage 2.

Stage 3: The sequence of `char`s accumulated in stage 2 (the field) is converted to a numeric value by the rules of one of the functions declared in the header `<cstdlib>`:

(3.1) — For a signed integer value, the function `strtoll`.

(3.2) — For an unsigned integer value, the function `strtoull`.

(3.3) — For a `float` value, the function `strtof`.

(3.4) — For a `double` value, the function `strtod`.

(3.5) — For a `long double` value, the function `strtold`.

The numeric value to be stored can be one of:

(3.6) — zero, if the conversion function does not convert the entire field.

(3.7) — the most positive (or negative) representable value, if the field to be converted to a signed integer type represents a value too large positive (or negative) to be represented in `val`.

(3.8) — the most positive representable value, if the field to be converted to an unsigned integer type represents a value that cannot be represented in `val`.

(3.9) — the converted value, otherwise.

The resultant numeric value is stored in `val`. If the conversion function does not convert the entire field, or if the field represents a value outside the range of representable values, `ios_base::failbit` is assigned to `err`.

4 Digit grouping is checked. That is, the positions of discarded separators is examined for consistency with `use_facet<numpunct<charT>>(loc).grouping()`. If they are not consistent then `ios_base::failbit` is assigned to `err`.

5 In any case, if stage 2 processing was terminated by the test for `in == end` then `err |= ios_base::eofbit` is performed.

```
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, bool& val) const;
```

6 *Effects:* If `(str.flags()&ios_base::boolalpha) == 0` then input proceeds as it would for a `long` except that if a value is being stored into `val`, the value is determined according to the following: If the value to be stored is 0 then `false` is stored. If the value is `1` then `true` is stored. Otherwise `true` is stored and `ios_base::failbit` is assigned to `err`.

7 Otherwise target sequences are determined "as if" by calling the members `falsename()` and `truename()` of the facet obtained by `use_facet<numpunct<charT>>(str.getloc())`. Successive characters in the range `[in, end)` (see **??**) are obtained and matched against corresponding positions in the target sequences only as necessary to identify a unique match. The input iterator `in` is compared to `end` only when necessary to obtain a character. If a target sequence is uniquely matched, `val` is set to the corresponding value. Otherwise `false` is stored and `ios_base::failbit` is assigned to `err`.

8 The `in` iterator is always left pointing one position beyond the last character successfully matched. If `val` is set, then `err` is set to `str.goodbit`; or to `str.eofbit` if, when seeking another character to match, it is found that `(in == end)`. If `val` is not set, then `err` is set to `str.failbit`; or to `(str.failbit|str.eofbit)` if the reason for the failure was that `(in == end)`. [*Example*: For targets `true`: `"a"` and `false`: `"abb"`, the input sequence `"a"` yields `val == true` and `err == str.eofbit`; the input sequence `"abc"` yields `err = str.failbit`, with `in` ending at the `'c'` element. For targets `true`: `"1"` and `false`: `"0"`, the input sequence `"1"` yields `val == true` and `err == str.goodbit`. For empty targets `("")`, any input sequence yields `err == str.failbit`. — *end example*]

9 *Returns:* `in`.

28.4.2.2 Class template `num_put` [locale.nm.put]

```
namespace std {
  template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class num_put : public locale::facet {
    public:
      using char_type = charT;
      using iter_type = OutputIterator;
```

```
    explicit num_put(size_t refs = 0);

    iter_type put(iter_type s, ios_base& f, char_type fill, bool v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, long long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, unsigned long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, unsigned long long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, double v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, long double v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, const void* v) const;

    static locale::id id;

  protected:
    ~num_put();
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, bool v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, long v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, long long v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, unsigned long) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, unsigned long long) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, double v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, long double v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill, const void* v) const;
  };
}
```

1 The facet `num_put` is used to format numeric values to a character sequence such as an ostream.

28.4.2.2.1 Members [facet.num.put.members]

```
iter_type put(iter_type out, ios_base& str, char_type fill, bool val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, long long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, unsigned long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, unsigned long long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, double val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, long double val) const;
iter_type put(iter_type out, ios_base& str, char_type fill, const void* val) const;
```

1 *Returns:* `do_put(out, str, fill, val)`.

28.4.2.2.2 Virtual functions [facet.num.put.virtuals]

```
iter_type do_put(iter_type out, ios_base& str, char_type fill, long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill, long long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill, unsigned long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill, unsigned long long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill, double val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill, long double val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill, const void* val) const;
```

1 *Effects:* Writes characters to the sequence `out`, formatting `val` as desired. In the following description, `loc` names a local variable initialized as

```
locale loc = str.getloc();
```

2 The details of this operation occur in several stages:

(2.1) — Stage 1: Determine a printf conversion specifier `spec` and determine the characters that would be printed by `printf` (??) given this conversion specifier for

```
printf(spec, val)
```

assuming that the current locale is the `"C"` locale.

(2.2) — Stage 2: Adjust the representation by converting each `char` determined by stage 1 to a `charT` using a conversion and values returned by members of `use_facet<numpunct<charT>>(loc)`

(2.3) — Stage 3: Determine where padding is required.

(2.4) — Stage 4: Insert the sequence into the `out`.

3 Detailed descriptions of each stage follow.

4 *Returns:* `out`.

5

Stage 1: The first action of stage 1 is to determine a conversion specifier. The tables that describe this determination use the following local variables

```
fmtflags flags = str.flags();
fmtflags basefield =  (flags & (ios_base::basefield));
fmtflags uppercase =  (flags & (ios_base::uppercase));
fmtflags floatfield = (flags & (ios_base::floatfield));
fmtflags showpos =    (flags & (ios_base::showpos));
fmtflags showbase =   (flags & (ios_base::showbase));
fmtflags showpoint =  (flags & (ios_base::showpoint));
```

All tables used in describing stage 1 are ordered. That is, the first line whose condition is true applies. A line without a condition is the default behavior when none of the earlier lines apply.

For conversion from an integral type other than a character type, the function determines the integral conversion specifier as indicated in Table 107.

Table 107: Integer conversions [tab:facet.num.put.int]

State	`stdio` equivalent
`basefield == ios_base::oct`	`%o`
`(basefield == ios_base::hex) && !uppercase`	`%x`
`(basefield == ios_base::hex)`	`%X`
for a `signed` integral type	`%d`
for an `unsigned` integral type	`%u`

For conversion from a floating-point type, the function determines the floating-point conversion specifier as indicated in Table 108.

Table 108: Floating-point conversions [tab:facet.num.put.fp]

State	`stdio` equivalent
`floatfield == ios_base::fixed`	`%f`
`floatfield == ios_base::scientific && !uppercase`	`%e`
`floatfield == ios_base::scientific`	`%E`
`floatfield == (ios_base::fixed | ios_base::scientific) && !uppercase`	`%a`
`floatfield == (ios_base::fixed | ios_base::scientific)`	`%A`
`!uppercase`	`%g`
otherwise	`%G`

For conversions from an integral or floating-point type a length modifier is added to the conversion specifier as indicated in Table 109.

Table 109: Length modifier [tab:facet.num.put.length]

Type	Length modifier
`long`	`l`
`long long`	`ll`
`unsigned long`	`l`
`unsigned long long`	`ll`
`long double`	`L`
otherwise	*none*

The conversion specifier has the following optional additional qualifiers prepended as indicated in Table 110.

Table 110: Numeric conversions [tab:facet.num.put.conv]

Type(s)	State	stdio equivalent
an integral type	`showpos`	`+`
	`showbase`	`#`
a floating-point type	`showpos`	`+`
	`showpoint`	`#`

For conversion from a floating-point type, if `floatfield != (ios_base::fixed | ios_base::scientific)`, `str.precision()` is specified as precision in the conversion specification. Otherwise, no precision is specified.

For conversion from `void*` the specifier is `%p`.

The representations at the end of stage 1 consists of the `char`'s that would be printed by a call of `printf(s, val)` where `s` is the conversion specifier determined above.

Stage 2: Any character `c` other than a decimal point(.) is converted to a `charT` via

```
use_facet<ctype<charT>>(loc).widen(c)
```

A local variable `punct` is initialized via

```
const numpunct<charT>& punct = use_facet<numpunct<charT>>(loc);
```

For arithmetic types, `punct.thousands_sep()` characters are inserted into the sequence as determined by the value returned by `punct.do_grouping()` using the method described in 28.4.3.1.2

Decimal point characters(.) are replaced by `punct.decimal_point()`

Stage 3: A local variable is initialized as

```
fmtflags adjustfield = (flags & (ios_base::adjustfield));
```

The location of any padding[268] is determined according to Table 111.

Table 111: Fill padding [tab:facet.num.put.fill]

State	Location
`adjustfield == ios_base::left`	pad after
`adjustfield == ios_base::right`	pad before
`adjustfield == internal` and a sign occurs in the representation	pad after the sign
`adjustfield == internal` and representation after stage 1 began with 0x or 0X	pad after x or X
otherwise	pad before

If `str.width()` is nonzero and the number of `charT`'s in the sequence after stage 2 is less than `str.width()`, then enough `fill` characters are added to the sequence at the position indicated for padding to bring the length of the sequence to `str.width()`.

`str.width(0)` is called.

Stage 4: The sequence of `charT`'s at the end of stage 3 are output via

```
*out++ = c
```

```
iter_type do_put(iter_type out, ios_base& str, char_type fill, bool val) const;
```

6 *Returns:* If `(str.flags() & ios_base::boolalpha) == 0` returns `do_put(out, str, fill, (int)val)`, otherwise obtains a string `s` as if by

```
string_type s =
  val ? use_facet<numpunct<charT>>(loc).truename()
      : use_facet<numpunct<charT>>(loc).falsename();
```

and then inserts each character `c` of `s` into `out` via `*out++ = c` and returns `out`.

268) The conversion specification `#o` generates a leading `0` which is *not* a padding character.

28.4.3 The numeric punctuation facet [facet.numpunct]

28.4.3.1 Class template numpunct [locale.numpunct]

```
namespace std {
  template<class charT>
    class numpunct : public locale::facet {
    public:
      using char_type   = charT;
      using string_type = basic_string<charT>;

      explicit numpunct(size_t refs = 0);

      char_type    decimal_point()   const;
      char_type    thousands_sep()   const;
      string       grouping()        const;
      string_type  truename()        const;
      string_type  falsename()       const;

      static locale::id id;

    protected:
      ~numpunct();                // virtual
      virtual char_type    do_decimal_point() const;
      virtual char_type    do_thousands_sep() const;
      virtual string       do_grouping()      const;
      virtual string_type  do_truename()      const;      // for bool
      virtual string_type  do_falsename()     const;      // for bool
    };
}
```

1 `numpunct<>` specifies numeric punctuation. The specializations required in Table 101 (28.3.1.1.1), namely `numpunct<wchar_t>` and `numpunct<char>`, provide classic `"C"` numeric formats, i.e., they contain information equivalent to that contained in the `"C"` locale or their wide character counterparts as if obtained by a call to `widen`.

2 The syntax for number formats is as follows, where `digit` represents the radix set specified by the `fmtflags` argument value, and `thousands-sep` and `decimal-point` are the results of corresponding `numpunct<charT>` members. Integer values have the format:

```
integer   ::= [sign] units
sign      ::= plusminus
plusminus ::= '+' | '-'
units     ::= digits [thousands-sep units]
digits    ::= digit [digits]
```

and floating-point values have:

```
floatval ::= [sign] units [decimal-point [digits]] [e [sign] digits] |
             [sign]       decimal-point  digits   [e [sign] digits]
e        ::= 'e' | 'E'
```

where the number of digits between `thousands-sep`s is as specified by `do_grouping()`. For parsing, if the `digits` portion contains no thousands-separators, no grouping constraint is applied.

28.4.3.1.1 Members [facet.numpunct.members]

`char_type decimal_point() const;`

1 *Returns:* `do_decimal_point()`.

`char_type thousands_sep() const;`

2 *Returns:* `do_thousands_sep()`.

`string grouping()  const;`

3 *Returns:* `do_grouping()`.

```
string_type truename()  const;
string_type falsename() const;
```

4 *Returns:* `do_truename()` or `do_falsename()`, respectively.

28.4.3.1.2 Virtual functions [facet.numpunct.virtuals]

```
char_type do_decimal_point() const;
```

1 *Returns:* A character for use as the decimal radix separator. The required specializations return `'.'` or `L'.'`.

```
char_type do_thousands_sep() const;
```

2 *Returns:* A character for use as the digit group separator. The required specializations return `','` or `L','`.

```
string do_grouping() const;
```

3 *Returns:* A ~~basic_string<char>~~`string` `vec` used as a vector of integer values, in which each element `vec[i]` represents the number of digits[269] in the group at position `i`, starting with position 0 as the rightmost group. If `vec.size() <= i`, the number is the same as group `(i - 1)`; if `(i < 0 || vec[i] <= 0 || vec[i] == CHAR_MAX)`, the size of the digit group is unlimited.

4 The required specializations return the empty string, indicating no grouping.

```
string_type do_truename()  const;
string_type do_falsename() const;
```

5 *Returns:* A string representing the name of the boolean value `true` or `false`, respectively.

6 In the base class implementation these names are `"true"` and `"false"`, or `L"true"` and `L"false"`.

28.4.3.2 Class template numpunct_byname [locale.numpunct.byname]

```
namespace std {
  template<class charT>
    class numpunct_byname : public numpunct<charT> {
    // this class is specialized for char and wchar_t.
    public:
      using char_type   = charT;
      using string_type = basic_string<charT>;

      explicit numpunct_byname(const char*, size_t refs = 0);
      explicit numpunct_byname(const string&, size_t refs = 0);

    protected:
      ~numpunct_byname();
    };
}
```

28.4.4 The collate category [category.collate]

28.4.4.1 Class template collate [locale.collate]

```
namespace std {
  template<class charT>
    class collate : public locale::facet {
    public:
      using char_type   = charT;
      using string_type = basic_string<charT>;

      explicit collate(size_t refs = 0);

      int compare(const charT* low1, const charT* high1,
                  const charT* low2, const charT* high2) const;
      string_type transform(const charT* low, const charT* high) const;
      long hash(const charT* low, const charT* high) const;
```

269) Thus, the string `"\003"` specifies groups of 3 digits each, and `"3"` probably indicates groups of 51 (!) digits each, because 51 is the ASCII value of `"3"`.

```
    static locale::id id;

  protected:
    ~collate();
    virtual int do_compare(const charT* low1, const charT* high1,
                           const charT* low2, const charT* high2) const;
    virtual string_type do_transform(const charT* low, const charT* high) const;
    virtual long do_hash (const charT* low, const charT* high) const;
  };
}
```

1 The class `collate<charT>` provides features for use in the collation (comparison) and hashing of strings. A locale member function template, `operator()`, uses the collate facet to allow a locale to act directly as the predicate argument for standard algorithms (**??**) and containers operating on strings. The specializations required in Table 101 (28.3.1.1.1), namely `collate<char>` and `collate<wchar_t>`, apply lexicographic ordering (**??**).

2 Each function compares a string of characters `*p` in the range `[low, high)`.

28.4.4.1.1 Members [locale.collate.members]

```
int compare(const charT* low1, const charT* high1,
            const charT* low2, const charT* high2) const;
```

1 *Returns:* `do_compare(low1, high1, low2, high2)`.

```
string_type transform(const charT* low, const charT* high) const;
```

2 *Returns:* `do_transform(low, high)`.

```
long hash(const charT* low, const charT* high) const;
```

3 *Returns:* `do_hash(low, high)`.

28.4.4.1.2 Virtual functions [locale.collate.virtuals]

```
int do_compare(const charT* low1, const charT* high1,
               const charT* low2, const charT* high2) const;
```

1 *Returns:* `1` if the first string is greater than the second, `-1` if less, zero otherwise. The specializations required in Table 101 (28.3.1.1.1), namely `collate<char>` and `collate<wchar_t>`, implement a lexicographical comparison (**??**).

```
string_type do_transform(const charT* low, const charT* high) const;
```

2 *Returns:* A `basic_string<charT>` value that, compared lexicographically with the result of calling `transform()` on another string, yields the same result as calling `do_compare()` on the same two strings.[270]

```
long do_hash(const charT* low, const charT* high) const;
```

3 *Returns:* An integer value equal to the result of calling `hash()` on any other string for which `do_compare()` returns 0 (equal) when passed the two strings. [*Note*: The probability that the result equals that for another string which does not compare equal should be very small, approaching `(1.0/numeric_limits<unsigned long>::max())`. — *end note*]

28.4.4.2 Class template `collate_byname` [locale.collate.byname]

```
namespace std {
  template<class charT>
    class collate_byname : public collate<charT> {
    public:
      using string_type = basic_string<charT>;

      explicit collate_byname(const char*, size_t refs = 0);
      explicit collate_byname(const string&, size_t refs = 0);
```

270) This function is useful when one string is being compared to many other strings.

```
  protected:
    ~collate_byname();
  };
}
```

28.4.5 The time category [category.time]

1 Templates `time_get<charT, InputIterator>` and `time_put<charT, OutputIterator>` provide date and time formatting and parsing. All specifications of member functions for `time_put` and `time_get` in the subclauses of 28.4.5 only apply to the specializations required in Tables 101 and 102 (28.3.1.1.1). Their members use their `ios_base&`, `ios_base::iostate&`, and `fill` arguments as described in 28.4, and the `ctype<>` facet, to determine formatting details.

28.4.5.1 Class template `time_get` [locale.time.get]

```
namespace std {
  class time_base {
  public:
    enum dateorder { no_order, dmy, mdy, ymd, ydm };
  };

  template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class time_get : public locale::facet, public time_base {
    public:
      using char_type = charT;
      using iter_type = InputIterator;

      explicit time_get(size_t refs = 0);

      dateorder date_order()  const { return do_date_order(); }
      iter_type get_time(iter_type s, iter_type end, ios_base& f,
                         ios_base::iostate& err, tm* t)  const;
      iter_type get_date(iter_type s, iter_type end, ios_base& f,
                         ios_base::iostate& err, tm* t)  const;
      iter_type get_weekday(iter_type s, iter_type end, ios_base& f,
                         ios_base::iostate& err, tm* t) const;
      iter_type get_monthname(iter_type s, iter_type end, ios_base& f,
                         ios_base::iostate& err, tm* t) const;
      iter_type get_year(iter_type s, iter_type end, ios_base& f,
                         ios_base::iostate& err, tm* t) const;
      iter_type get(iter_type s, iter_type end, ios_base& f,
                         ios_base::iostate& err, tm* t, char format, char modifier = 0) const;
      iter_type get(iter_type s, iter_type end, ios_base& f,
                         ios_base::iostate& err, tm* t, const char_type* fmt,
                         const char_type* fmtend) const;

      static locale::id id;

    protected:
      ~time_get();
      virtual dateorder do_date_order()  const;
      virtual iter_type do_get_time(iter_type s, iter_type end, ios_base&,
                                    ios_base::iostate& err, tm* t) const;
      virtual iter_type do_get_date(iter_type s, iter_type end, ios_base&,
                                    ios_base::iostate& err, tm* t) const;
      virtual iter_type do_get_weekday(iter_type s, iter_type end, ios_base&,
                                       ios_base::iostate& err, tm* t) const;
      virtual iter_type do_get_monthname(iter_type s, iter_type end, ios_base&,
                                         ios_base::iostate& err, tm* t) const;
      virtual iter_type do_get_year(iter_type s, iter_type end, ios_base&,
                                    ios_base::iostate& err, tm* t) const;
      virtual iter_type do_get(iter_type s, iter_type end, ios_base& f,
                               ios_base::iostate& err, tm* t, char format, char modifier) const;
    };
}
```

1 `time_get` is used to parse a character sequence, extracting components of a time or date into a `struct tm` object. Each `get` member parses a format as produced by a corresponding format specifier to `time_put<>::put`. If the sequence being parsed matches the correct format, the corresponding members of the `struct tm` argument are set to the values used to produce the sequence; otherwise either an error is reported or unspecified values are assigned.[271]

2 If the end iterator is reached during parsing by any of the `get()` member functions, the member sets `ios_base::eofbit` in `err`.

28.4.5.1.1 Members [locale.time.get.members]

```
dateorder date_order() const;
```

1 *Returns:* `do_date_order()`.

```
iter_type get_time(iter_type s, iter_type end, ios_base& str,
                   ios_base::iostate& err, tm* t) const;
```

2 *Returns:* `do_get_time(s, end, str, err, t)`.

```
iter_type get_date(iter_type s, iter_type end, ios_base& str,
                   ios_base::iostate& err, tm* t) const;
```

3 *Returns:* `do_get_date(s, end, str, err, t)`.

```
iter_type get_weekday(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
iter_type get_monthname(iter_type s, iter_type end, ios_base& str,
                        ios_base::iostate& err, tm* t) const;
```

4 *Returns:* `do_get_weekday(s, end, str, err, t)` or `do_get_monthname(s, end, str, err, t)`.

```
iter_type get_year(iter_type s, iter_type end, ios_base& str,
                   ios_base::iostate& err, tm* t) const;
```

5 *Returns:* `do_get_year(s, end, str, err, t)`.

```
iter_type get(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err,
              tm* t, char format, char modifier = 0) const;
```

6 *Returns:* `do_get(s, end, f, err, t, format, modifier)`.

```
iter_type get(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err,
              tm* t, const char_type* fmt, const char_type* fmtend) const;
```

7 ~~*Requires:*~~ *Expects:* `[fmt, fmtend)` ~~shall be~~is a valid range.

8 *Effects:* The function starts by evaluating `err = ios_base::goodbit`. It then enters a loop, reading zero or more characters from `s` at each iteration. Unless otherwise specified below, the loop terminates when the first of the following conditions holds:

(8.1) — The expression `fmt == fmtend` evaluates to `true`.

(8.2) — The expression `err == ios_base::goodbit` evaluates to `false`.

(8.3) — The expression `s == end` evaluates to `true`, in which case the function evaluates `err = ios_base::eofbit | ios_base::failbit`.

(8.4) — The next element of `fmt` is equal to `'%'`, optionally followed by a modifier character, followed by a conversion specifier character, `format`, together forming a conversion specification valid for the ISO/IEC 9945 function `strptime`. If the number of elements in the range `[fmt, fmtend)` is not sufficient to unambiguously determine whether the conversion specification is complete and valid, the function evaluates `err = ios_base::failbit`. Otherwise, the function evaluates `s = do_get(s, end, f, err, t, format, modifier)`, where the value of `modifier` is `'\0'` when the optional modifier is absent from the conversion specification. If `err == ios_base::goodbit` holds after the evaluation of the expression, the function increments `fmt` to point just past the end of the conversion specification and continues looping.

271) In other words, user confirmation is required for reliable parsing of user-entered dates and times, but machine-generated formats can be parsed reliably. This allows parsers to be aggressive about interpreting user variations on standard formats.

(8.5) — The expression `isspace(*fmt, f.getloc())` evaluates to `true`, in which case the function first increments `fmt` until `fmt == fmtend || !isspace(*fmt, f.getloc())` evaluates to `true`, then advances `s` until `s == end || !isspace(*s, f.getloc())` is `true`, and finally resumes looping.

(8.6) — The next character read from `s` matches the element pointed to by `fmt` in a case-insensitive comparison, in which case the function evaluates `++fmt, ++s` and continues looping. Otherwise, the function evaluates `err = ios_base::failbit`.

9 [*Note*: The function uses the `ctype<charT>` facet installed in `f`'s locale to determine valid whitespace characters. It is unspecified by what means the function performs case-insensitive comparison or whether multi-character sequences are considered while doing so. — *end note*]

10 *Returns:* `s`.

28.4.5.1.2 Virtual functions [locale.time.get.virtuals]

```
dateorder do_date_order() const;
```

1 *Returns:* An enumeration value indicating the preferred order of components for those date formats that are composed of day, month, and year.[272] Returns `no_order` if the date format specified by `'x'` contains other variable components (e.g., Julian day, week number, week day).

```
iter_type do_get_time(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
```

2 *Effects:* Reads characters starting at `s` until it has extracted those `struct tm` members, and remaining format characters, used by `time_put<>::put` to produce the format specified by `"%H:%M:%S"`, or until it encounters an error or end of sequence.

3 *Returns:* An iterator pointing immediately beyond the last character recognized as possibly part of a valid time.

```
iter_type do_get_date(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
```

4 *Effects:* Reads characters starting at `s` until it has extracted those `struct tm` members and remaining format characters used by `time_put<>::put` to produce one of the following formats, or until it encounters an error. The format depends on the value returned by `date_order()` as shown in Table 112.

Table 112: `do_get_date` effects [tab:locale.time.get.dogetdate]

`date_order()`	Format
`no_order`	`"%m%d%y"`
`dmy`	`"%d%m%y"`
`mdy`	`"%m%d%y"`
`ymd`	`"%y%m%d"`
`ydm`	`"%y%d%m"`

5 An implementation may also accept additional implementation-defined formats.

6 *Returns:* An iterator pointing immediately beyond the last character recognized as possibly part of a valid date.

```
iter_type do_get_weekday(iter_type s, iter_type end, ios_base& str,
                         ios_base::iostate& err, tm* t) const;
iter_type do_get_monthname(iter_type s, iter_type end, ios_base& str,
                           ios_base::iostate& err, tm* t) const;
```

7 *Effects:* Reads characters starting at `s` until it has extracted the (perhaps abbreviated) name of a weekday or month. If it finds an abbreviation that is followed by characters that could match a full name, it continues reading until it matches the full name or fails. It sets the appropriate `struct tm` member accordingly.

8 *Returns:* An iterator pointing immediately beyond the last character recognized as part of a valid name.

272) This function is intended as a convenience only, for common formats, and may return `no_order` in valid locales.

```
iter_type do_get_year(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
```

9 *Effects:* Reads characters starting at `s` until it has extracted an unambiguous year identifier. It is implementation-defined whether two-digit year numbers are accepted, and (if so) what century they are assumed to lie in. Sets the `t->tm_year` member accordingly.

10 *Returns:* An iterator pointing immediately beyond the last character recognized as part of a valid year identifier.

```
iter_type do_get(iter_type s, iter_type end, ios_base& f,
                 ios_base::iostate& err, tm* t, char format, char modifier) const;
```

11 ~~*Requires:*~~ *Expects:* `t` ~~shall point~~points to an object.

12 *Effects:* The function starts by evaluating `err = ios_base::goodbit`. It then reads characters starting at `s` until it encounters an error, or until it has extracted and assigned those `struct tm` members, and any remaining format characters, corresponding to a conversion directive appropriate for the ISO/IEC 9945 function `strptime`, formed by concatenating `'%'`, the `modifier` character, when non-NUL, and the `format` character. When the concatenation fails to yield a complete valid directive the function leaves the object pointed to by `t` unchanged and evaluates `err |= ios_base::failbit`. When `s == end` evaluates to `true` after reading a character the function evaluates `err |= ios_base::eofbit`.

13 For complex conversion directives such as `%c`, `%x`, or `%X`, or directives that involve the optional modifiers `E` or `O`, when the function is unable to unambiguously determine some or all `struct tm` members from the input sequence `[s, end)`, it evaluates `err |= ios_base::eofbit`. In such cases the values of those `struct tm` members are unspecified and may be outside their valid range.

14 *Remarks:* It is unspecified whether multiple calls to `do_get()` with the address of the same `struct tm` object will update the current contents of the object or simply overwrite its members. Portable programs should zero out the object before invoking the function.

15 *Returns:* An iterator pointing immediately beyond the last character recognized as possibly part of a valid input sequence for the given `format` and `modifier`.

28.4.5.2 Class template `time_get_byname` [locale.time.get.byname]

```
namespace std {
  template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class time_get_byname : public time_get<charT, InputIterator> {
    public:
      using dateorder = time_base::dateorder;
      using iter_type = InputIterator;

      explicit time_get_byname(const char*, size_t refs = 0);
      explicit time_get_byname(const string&, size_t refs = 0);

    protected:
      ~time_get_byname();
    };
}
```

28.4.5.3 Class template `time_put` [locale.time.put]

```
namespace std {
  template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class time_put : public locale::facet {
    public:
      using char_type = charT;
      using iter_type = OutputIterator;

      explicit time_put(size_t refs = 0);

      // the following is implemented in terms of other member functions.
      iter_type put(iter_type s, ios_base& f, char_type fill, const tm* tmb,
                    const charT* pattern, const charT* pat_end) const;
      iter_type put(iter_type s, ios_base& f, char_type fill,
                    const tm* tmb, char format, char modifier = 0) const;
```

```
    static locale::id id;

  protected:
    ~time_put();
    virtual iter_type do_put(iter_type s, ios_base&, char_type, const tm* t,
                             char format, char modifier) const;
  };
}
```

28.4.5.3.1 Members [locale.time.put.members]

```
iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
              const charT* pattern, const charT* pat_end) const;
iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
              char format, char modifier = 0) const;
```

1 *Effects:* The first form steps through the sequence from `pattern` to `pat_end`, identifying characters that are part of a format sequence. Each character that is not part of a format sequence is written to `s` immediately, and each format sequence, as it is identified, results in a call to `do_put`; thus, format elements and other characters are interleaved in the output in the order in which they appear in the pattern. Format sequences are identified by converting each character `c` to a `char` value as if by `ct.narrow(c, 0)`, where `ct` is a reference to `ctype<charT>` obtained from `str.getloc()`. The first character of each sequence is equal to `'%'`, followed by an optional modifier character `mod`[273] and a format specifier character `spec` as defined for the function `strftime`. If no modifier character is present, `mod` is zero. For each valid format sequence identified, calls `do_put(s, str, fill, t, spec, mod)`.

2 The second form calls `do_put(s, str, fill, t, format, modifier)`.

3 [*Note*: The `fill` argument may be used in the implementation-defined formats or by derivations. A space character is a reasonable default for this argument. — *end note*]

4 *Returns:* An iterator pointing immediately after the last character produced.

28.4.5.3.2 Virtual functions [locale.time.put.virtuals]

```
iter_type do_put(iter_type s, ios_base&, char_type fill, const tm* t,
                 char format, char modifier) const;
```

1 *Effects:* Formats the contents of the parameter `t` into characters placed on the output sequence `s`. Formatting is controlled by the parameters `format` and `modifier`, interpreted identically as the format specifiers in the string argument to the standard library function `strftime()`[274], except that the sequence of characters produced for those specifiers that are described as depending on the C locale are instead implementation-defined.[275]

2 *Returns:* An iterator pointing immediately after the last character produced. [*Note*: The `fill` argument may be used in the implementation-defined formats or by derivations. A space character is a reasonable default for this argument. — *end note*]

28.4.5.4 Class template `time_put_byname` [locale.time.put.byname]

```
namespace std {
  template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class time_put_byname : public time_put<charT, OutputIterator> {
    public:
      using char_type = charT;
      using iter_type = OutputIterator;

      explicit time_put_byname(const char*, size_t refs = 0);
      explicit time_put_byname(const string&, size_t refs = 0);

    protected:
      ~time_put_byname();
    };
}
```

273) Although the C programming language defines no modifiers, most vendors do.
274) Interpretation of the `modifier` argument is implementation-defined, but should follow POSIX conventions.
275) Implementations should refer to other standards such as POSIX for these definitions.

28.4.6 The monetary category [category.monetary]

1 These templates handle monetary formats. A template parameter indicates whether local or international monetary formats are to be used.

2 All specifications of member functions for `money_put` and `money_get` in the subclauses of 28.4.6 only apply to the specializations required in Tables 101 and 102 (28.3.1.1.1). Their members use their `ios_base&`, `ios_base::iostate&`, and `fill` arguments as described in 28.4, and the `moneypunct<>` and `ctype<>` facets, to determine formatting details.

28.4.6.1 Class template `money_get` [locale.money.get]

```
namespace std {
  template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class money_get : public locale::facet {
    public:
      using char_type   = charT;
      using iter_type   = InputIterator;
      using string_type = basic_string<charT>;

      explicit money_get(size_t refs = 0);

      iter_type get(iter_type s, iter_type end, bool intl,
                    ios_base& f, ios_base::iostate& err,
                    long double& units) const;
      iter_type get(iter_type s, iter_type end, bool intl,
                    ios_base& f, ios_base::iostate& err,
                    string_type& digits) const;

      static locale::id id;

    protected:
      ~money_get();
      virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
                               ios_base::iostate& err, long double& units) const;
      virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
                               ios_base::iostate& err, string_type& digits) const;
    };
}
```

28.4.6.1.1 Members [locale.money.get.members]

```
iter_type get(iter_type s, iter_type end, bool intl, ios_base& f,
              ios_base::iostate& err, long double& quant) const;
iter_type get(iter_type s, iter_type end, bool intl, ios_base& f,
              ios_base::iostate& err, string_type& quant) const;
```

1 *Returns:* `do_get(s, end, intl, f, err, quant)`.

28.4.6.1.2 Virtual functions [locale.money.get.virtuals]

```
iter_type do_get(iter_type s, iter_type end, bool intl, ios_base& str,
                 ios_base::iostate& err, long double& units) const;
iter_type do_get(iter_type s, iter_type end, bool intl, ios_base& str,
                 ios_base::iostate& err, string_type& digits) const;
```

1 *Effects:* Reads characters from `s` to parse and construct a monetary value according to the format specified by a `moneypunct<charT, Intl>` facet reference `mp` and the character mapping specified by a `ctype<charT>` facet reference `ct` obtained from the locale returned by `str.getloc()`, and `str.flags()`. If a valid sequence is recognized, does not change `err`; otherwise, sets `err` to `(err|str.failbit)`, or `(err|str.failbit|str.eofbit)` if no more characters are available, and does not change `units` or `digits`. Uses the pattern returned by `mp.neg_format()` to parse all values. The result is returned as an integral value stored in `units` or as a sequence of digits possibly preceded by a minus sign (as produced by `ct.widen(c)` where `c` is `'-'` or in the range from `'0'` through `'9'` (inclusive)) stored in `digits`. [*Example*: The sequence `$1,056.23` in a common United States locale would yield, for `units`, `105623`, or, for `digits`, `"105623"`. — *end example*] If `mp.grouping()` indicates that no thousands

separators are permitted, any such characters are not read, and parsing is terminated at the point where they first appear. Otherwise, thousands separators are optional; if present, they are checked for correct placement only after all format components have been read.

2 Where `money_base::space` or `money_base::none` appears as the last element in the format pattern, no white space is consumed. Otherwise, where `money_base::space` appears in any of the initial elements of the format pattern, at least one white space character is required. Where `money_base::none` appears in any of the initial elements of the format pattern, white space is allowed but not required. If `(str.flags() & str.showbase)` is false, the currency symbol is optional and is consumed only if other characters are needed to complete the format; otherwise, the currency symbol is required.

3 If the first character (if any) in the string `pos` returned by `mp.positive_sign()` or the string `neg` returned by `mp.negative_sign()` is recognized in the position indicated by `sign` in the format pattern, it is consumed and any remaining characters in the string are required after all the other format components. [*Example*: If `showbase` is off, then for a `neg` value of `"()"` and a currency symbol of `"L"`, in `"(100 L)"` the `"L"` is consumed; but if `neg` is `"-"`, the `"L"` in `"-100 L"` is not consumed. — *end example*] If `pos` or `neg` is empty, the sign component is optional, and if no sign is detected, the result is given the sign that corresponds to the source of the empty string. Otherwise, the character in the indicated position must match the first character of `pos` or `neg`, and the result is given the corresponding sign. If the first character of `pos` is equal to the first character of `neg`, or if both strings are empty, the result is given a positive sign.

4 Digits in the numeric monetary component are extracted and placed in `digits`, or into a character buffer `buf1` for conversion to produce a value for `units`, in the order in which they appear, preceded by a minus sign if and only if the result is negative. The value `units` is produced as if by[276]

```
for (int i = 0; i < n; ++i)
  buf2[i] = src[find(atoms, atoms+sizeof(src), buf1[i]) - atoms];
buf2[n] = 0;
sscanf(buf2, "%Lf", &units);
```

where `n` is the number of characters placed in `buf1`, `buf2` is a character buffer, and the values `src` and `atoms` are defined as if by

```
static const char src[] = "0123456789-";
charT atoms[sizeof(src)];
ct.widen(src, src + sizeof(src) - 1, atoms);
```

5 *Returns:* An iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

28.4.6.2 Class template `money_put` [locale.money.put]

```
namespace std {
  template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class money_put : public locale::facet {
    public:
      using char_type   = charT;
      using iter_type   = OutputIterator;
      using string_type = basic_string<charT>;

      explicit money_put(size_t refs = 0);

      iter_type put(iter_type s, bool intl, ios_base& f,
                    char_type fill, long double units) const;
      iter_type put(iter_type s, bool intl, ios_base& f,
                    char_type fill, const string_type& digits) const;

      static locale::id id;

    protected:
      ~money_put();
      virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
                               long double units) const;
```

276) The semantics here are different from `ct.narrow`.

```
    virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
                             const string_type& digits) const;
  };
}
```

28.4.6.2.1 Members [locale.money.put.members]

```
iter_type put(iter_type s, bool intl, ios_base& f, char_type fill, long double quant) const;
iter_type put(iter_type s, bool intl, ios_base& f, char_type fill, const string_type& quant) const;
```

1 *Returns:* `do_put(s, intl, f, loc, quant)`.

28.4.6.2.2 Virtual functions [locale.money.put.virtuals]

```
iter_type do_put(iter_type s, bool intl, ios_base& str,
                 char_type fill, long double units) const;
iter_type do_put(iter_type s, bool intl, ios_base& str,
                 char_type fill, const string_type& digits) const;
```

1 *Effects:* Writes characters to `s` according to the format specified by a `moneypunct<charT, Intl>` facet reference `mp` and the character mapping specified by a `ctype<charT>` facet reference `ct` obtained from the locale returned by `str.getloc()`, and `str.flags()`. The argument `units` is transformed into a sequence of wide characters as if by

```
ct.widen(buf1, buf1 + sprintf(buf1, "%.0Lf", units), buf2)
```

for character buffers `buf1` and `buf2`. If the first character in `digits` or `buf2` is equal to `ct.widen('-')`, then the pattern used for formatting is the result of `mp.neg_format()`; otherwise the pattern is the result of `mp.pos_format()`. Digit characters are written, interspersed with any thousands separators and decimal point specified by the format, in the order they appear (after the optional leading minus sign) in `digits` or `buf2`. In `digits`, only the optional leading minus sign and the immediately subsequent digit characters (as classified according to `ct`) are used; any trailing characters (including digits appearing after a non-digit character) are ignored. Calls `str.width(0)`.

2 *Remarks:* The currency symbol is generated if and only if `(str.flags() & str.showbase)` is nonzero. If the number of characters generated for the specified format is less than the value returned by `str.width()` on entry to the function, then copies of `fill` are inserted as necessary to pad to the specified width. For the value `af` equal to `(str.flags() & str.adjustfield)`, if `(af == str.internal)` is `true`, the fill characters are placed where `none` or `space` appears in the formatting pattern; otherwise if `(af == str.left)` is `true`, they are placed after the other characters; otherwise, they are placed before the other characters. [*Note*: It is possible, with some combinations of format patterns and flag values, to produce output that cannot be parsed using `num_get<>::get`. — *end note*]

3 *Returns:* An iterator pointing immediately after the last character produced.

28.4.6.3 Class template `moneypunct` [locale.moneypunct]

```
namespace std {
  class money_base {
  public:
    enum part { none, space, symbol, sign, value };
    struct pattern { char field[4]; };
  };

  template<class charT, bool International = false>
    class moneypunct : public locale::facet, public money_base {
    public:
      using char_type   = charT;
      using string_type = basic_string<charT>;

      explicit moneypunct(size_t refs = 0);

      charT        decimal_point() const;
      charT        thousands_sep() const;
      string       grouping()      const;
      string_type  curr_symbol()   const;
      string_type  positive_sign() const;
```

```
    string_type  negative_sign() const;
    int          frac_digits()   const;
    pattern      pos_format()    const;
    pattern      neg_format()    const;

    static locale::id id;
    static const bool intl = International;

  protected:
    ~moneypunct();
    virtual charT        do_decimal_point() const;
    virtual charT        do_thousands_sep() const;
    virtual string       do_grouping()      const;
    virtual string_type  do_curr_symbol()   const;
    virtual string_type  do_positive_sign() const;
    virtual string_type  do_negative_sign() const;
    virtual int          do_frac_digits()   const;
    virtual pattern      do_pos_format()    const;
    virtual pattern      do_neg_format()    const;
  };
}
```

1 The `moneypunct<>` facet defines monetary formatting parameters used by `money_get<>` and `money_put<>`. A monetary format is a sequence of four components, specified by a `pattern` value `p`, such that the `part` value `static_cast<part>(p.field[i])` determines the i^{th} component of the format[277] In the `field` member of a `pattern` object, each value `symbol`, `sign`, `value`, and either `space` or `none` appears exactly once. The value `none`, if present, is not first; the value `space`, if present, is neither first nor last.

2 Where `none` or `space` appears, white space is permitted in the format, except where `none` appears at the end, in which case no white space is permitted. The value `space` indicates that at least one space is required at that position. Where `symbol` appears, the sequence of characters returned by `curr_symbol()` is permitted, and can be required. Where `sign` appears, the first (if any) of the sequence of characters returned by `positive_sign()` or `negative_sign()` (respectively as the monetary value is non-negative or negative) is required. Any remaining characters of the sign sequence are required after all other format components. Where `value` appears, the absolute numeric monetary value is required.

3 The format of the numeric monetary value is a decimal number:

```
value ::= units [ decimal-point [ digits ]] |
  decimal-point digits
```

if `frac_digits()` returns a positive value, or

```
value ::= units
```

otherwise. The symbol `decimal-point` indicates the character returned by `decimal_point()`. The other symbols are defined as follows:

```
units ::= digits [ thousands-sep units ]
digits ::= adigit [ digits ]
```

In the syntax specification, the symbol `adigit` is any of the values `ct.widen(c)` for `c` in the range `'0'` through `'9'` (inclusive) and `ct` is a reference of type `const ctype<charT>&` obtained as described in the definitions of `money_get<>` and `money_put<>`. The symbol `thousands-sep` is the character returned by `thousands_sep()`. The space character used is the value `ct.widen(' ')`. White space characters are those characters `c` for which `ci.is(space, c)` returns `true`. The number of digits required after the decimal point (if any) is exactly the value returned by `frac_digits()`.

4 The placement of thousands-separator characters (if any) is determined by the value returned by `grouping()`, defined identically as the member `numpunct<>::do_grouping()`.

28.4.6.3.1 Members [locale.moneypunct.members]

```
charT        decimal_point() const;
charT        thousands_sep() const;
string       grouping()      const;
string_type  curr_symbol()   const;
```

277) An array of `char`, rather than an array of `part`, is specified for `pattern::field` purely for efficiency.

```
string_type  positive_sign() const;
string_type  negative_sign() const;
int          frac_digits()   const;
pattern      pos_format()    const;
pattern      neg_format()    const;
```

1 Each of these functions *F* returns the result of calling the corresponding virtual member function `do_F()`.

28.4.6.3.2 Virtual functions [locale.moneypunct.virtuals]

```
charT do_decimal_point() const;
```

1 *Returns:* The radix separator to use in case `do_frac_digits()` is greater than zero.[278]

```
charT do_thousands_sep() const;
```

2 *Returns:* The digit group separator to use in case `do_grouping()` specifies a digit grouping pattern.[279]

```
string do_grouping() const;
```

3 *Returns:* A pattern defined identically as, but not necessarily equal to, the result of `numpunct<charT>::do_grouping()`.[280]

```
string_type do_curr_symbol() const;
```

4 *Returns:* A string to use as the currency identifier symbol. [*Note*: For specializations where the second template parameter is `true`, this is typically four characters long: a three-letter code as specified by ISO 4217 followed by a space. — *end note*]

```
string_type do_positive_sign() const;
string_type do_negative_sign() const;
```

5 *Returns:* `do_positive_sign()` returns the string to use to indicate a positive monetary value;[281] `do_negative_sign()` returns the string to use to indicate a negative value.

```
int do_frac_digits() const;
```

6 *Returns:* The number of digits after the decimal radix separator, if any.[282]

```
pattern do_pos_format() const;
pattern do_neg_format() const;
```

7 *Returns:* The specializations required in Table 102 (28.3.1.1.1), namely

(7.1) — `moneypunct<char>`,

(7.2) — `moneypunct<wchar_t>`,

(7.3) — `moneypunct<char, true>`, and

(7.4) — `moneypunct<wchar_t, true>`,

return an object of type `pattern` initialized to `{ symbol, sign, none, value }`.[283]

28.4.6.4 Class template `moneypunct_byname` [locale.moneypunct.byname]

```
namespace std {
  template<class charT, bool Intl = false>
  class moneypunct_byname : public moneypunct<charT, Intl> {
    public:
      using pattern     = money_base::pattern;
      using string_type = basic_string<charT>;

      explicit moneypunct_byname(const char*, size_t refs = 0);
      explicit moneypunct_byname(const string&, size_t refs = 0);
```

278) In common U.S. locales this is '.'.
279) In common U.S. locales this is ','.
280) To specify grouping by 3s, the value is `"\003"` *not* `"3"`.
281) This is usually the empty string.
282) In common U.S. locales, this is 2.
283) Note that the international symbol returned by `do_curr_symbol()` usually contains a space, itself; for example, `"USD "`.

```
    protected:
      ~moneypunct_byname();
    };
}
```

28.4.7 The message retrieval category [category.messages]

1 Class `messages<charT>` implements retrieval of strings from message catalogs.

28.4.7.1 Class template messages [locale.messages]

```
namespace std {
  class messages_base {
  public:
    using catalog = unspecified signed integer type;
  };

  template<class charT>
    class messages : public locale::facet, public messages_base {
    public:
      using char_type   = charT;
      using string_type = basic_string<charT>;

      explicit messages(size_t refs = 0);

      catalog open(const basic_string<char>string& fn, const locale&) const;
      string_type get(catalog c, int set, int msgid,
                      const string_type& dfault) const;
      void close(catalog c) const;

      static locale::id id;

    protected:
      ~messages();
      virtual catalog do_open(const basic_string<char>string&, const locale&) const;
      virtual string_type do_get(catalog, int set, int msgid,
                                 const string_type& dfault) const;
      virtual void do_close(catalog) const;
    };
}
```

1 Values of type `messages_base::catalog` usable as arguments to members `get` and `close` can be obtained only by calling member `open`.

28.4.7.1.1 Members [locale.messages.members]

```
catalog open(const basic_string<char>string& name, const locale& loc) const;
```

1 *Returns:* `do_open(name, loc)`.

```
string_type get(catalog cat, int set, int msgid, const string_type& dfault) const;
```

2 *Returns:* `do_get(cat, set, msgid, dfault)`.

```
void close(catalog cat) const;
```

3 *Effects:* Calls `do_close(cat)`.

28.4.7.1.2 Virtual functions [locale.messages.virtuals]

```
catalog do_open(const basic_string<char>string& name, const locale& loc) const;
```

1 *Returns:* A value that may be passed to `get()` to retrieve a message from the message catalog identified by the string `name` according to an implementation-defined mapping. The result can be used until it is passed to `close()`.

2 Returns a value less than 0 if no such catalog can be opened.

3 *Remarks:* The locale argument `loc` is used for character set code conversion when retrieving messages, if needed.

```
string_type do_get(catalog cat, int set, int msgid, const string_type& dfault) const;
```

4 ~~*Requires:*~~ *Expects:* `cat` ~~shall be~~is a catalog obtained from `open()` and not yet closed.

5 *Returns:* A message identified by arguments `set`, `msgid`, and `dfault`, according to an implementation-defined mapping. If no such message can be found, returns `dfault`.

```
void do_close(catalog cat) const;
```

6 ~~*Requires:*~~ *Expects:* `cat` ~~shall be~~is a catalog obtained from `open()` and not yet closed.

7 *Effects:* Releases unspecified resources associated with `cat`.

8 *Remarks:* The limit on such resources, if any, is implementation-defined.

28.4.7.2 Class template `messages_byname` [locale.messages.byname]

```
namespace std {
  template<class charT>
    class messages_byname : public messages<charT> {
    public:
      using catalog     = messages_base::catalog;
      using string_type = basic_string<charT>;

      explicit messages_byname(const char*, size_t refs = 0);
      explicit messages_byname(const string&, size_t refs = 0);

    protected:
      ~messages_byname();
    };
}
```

28.5 C library locales [c.locales]

28.5.1 Header `<clocale>` synopsis [clocale.syn]

```
namespace std {
  struct lconv;

  char* setlocale(int category, const char* locale);
  lconv* localeconv();
}

#define NULL see ??
#define LC_ALL see below
#define LC_COLLATE see below
#define LC_CTYPE see below
#define LC_MONETARY see below
#define LC_NUMERIC see below
#define LC_TIME see below
```

1 The contents and meaning of the header `<clocale>` are the same as the C standard library header `<locale.h>`.

2 Calls to the function `setlocale` may introduce a data race (**??**) with other calls to `setlocale` or with calls to the functions listed in Table 113.

See also: ISO C 7.11

Table 113: Potential `setlocale` data races [tab:setlocale.data.races]

`fprintf`	`isprint`	`iswdigit`	`localeconv`	`tolower`
`fscanf`	`ispunct`	`iswgraph`	`mblen`	`toupper`
`isalnum`	`isspace`	`iswlower`	`mbstowcs`	`towlower`
`isalpha`	`isupper`	`iswprint`	`mbtowc`	`towupper`
`isblank`	`iswalnum`	`iswpunct`	`setlocale`	`wcscoll`
`iscntrl`	`iswalpha`	`iswspace`	`strcoll`	`wcstod`
`isdigit`	`iswblank`	`iswupper`	`strerror`	`wcstombs`
`isgraph`	`iswcntrl`	`iswxdigit`	`strtod`	`wcsxfrm`
`islower`	`iswctype`	`isxdigit`	`strxfrm`	`wctomb`