INTERNATIONALIZATION

Table of Contents

Section 1.

Internationalization

1.1 How to Read this Section

This section of the *User's Guide* deals with locales in the Standard C++ Library. Since the focus here is on concepts rather than details, you will want to consult the *Class Reference* for more complete information.

We begin the section with an introduction to internationalization in general. It is intended to explain why and how locales are useful for the benefit of readers with no experience in this area. Eventually it will include a reference for the standard facets, but not in this first version of the *User's Guide*. Hence, the section may look a bit unbalanced for the time being.

Following the introduction, we describe the facilities in C that are currently available for internationalizing software. Users with a background in C will want to understand how the C locale differs from the C++ locale. Some developers may even need to know how the two locales interact.

For their benefit, we then contrast the concept of the C++ locale with the C locale. We learn what a C++ locale is, what facets are, how locales are created and composed, and how facets are used, replaced, and defined. The standard facets are only briefly described here, but details are available in the *Class Reference*.

For the advanced user, we conclude the internationalization section with a rather complex example of a user-defined facet, which demonstrates how facets can be built and used in conjunction with iostreams.

1.2 Internationalization and Localization

Computer users all over the world prefer to interact with their systems using their own local languages and cultural conventions. As a developer aiming for high international acceptance of your products, you need to provide users the flexibility for modifying output conventions to comply with local requirements, such as different currency and numeric representations. You must also provide the capability for translating interfaces and messages without necessitating many different language versions of your software.

Two processes that enhance software for worldwide use are *internationalization* and *localization*. *Internationalization* is the process of building into software the potential for worldwide use. It is the result of efforts by programmers and software designers during software development.

Internationalization requires that developers consciously design and implement software for adaptation to various languages and cultural conventions, and avoid hard-coding elements that can be localized, like screen positions and file names. For example, developers should never embed in their code any messages, prompts, or other kind of displayed text, but rather store the messages externally, so they can be translated and exchanged. A developer of internationalized software should never assume specific conventions for formatting numeric or monetary values or for displaying date and time.

Localization is the process of actually adapting internationalized software to the needs of users in a particular geographical or cultural area. It includes translation of messages by software translators. It requires the creation and availability of appropriate tables containing relevant local data for use in a given system. This typically is the function of system administrators, who build facilities for these functions into their operating systems. Users of internationalized software are involved in the process of localization in that they select the local conventions they prefer.

The *Standard C++ Library* offers a number of classes that support internationalization of your programs. We will describe them in detail in this chapter. Before we do, however, we would like to define some of the cultural conventions that impact software internationalization, and are supported by the programming languages C and C++ and their respective standard libraries. Of course, there are many issues outside our list that need to be addressed, like orientation, sizing and positioning of screen displays, vertical writing and printing, selection of font tables, handling international keyboards, and so on. But let us begin here.

1.2.1 Localizing Cultural Conventions

The need for localizing software arises from differences in cultural conventions. These differences involve: language itself;

representation of numbers and currency; display of time and date; and ordering or sorting of characters and strings.

1.2.1.1 Language

Of course, *language* itself varies from country to country, and even within a country. Your program may require output messages in English, Dutch, French, Italian or any number of languages commonly used in the world today.

Languages may also differ in the *alphabet* they use. Examples of different languages with their respective alphabets are given below:

American English:	`a-z A-Z` and punctuation
German:	`a-z A-Z` and punctuation and `äöü ÄÖÜ ß`
Greek:	α–ω· Α–Ω and punctuation

1.2.1.2 Numbers

The representation of *numbers* depends on local customs, which vary from country to country. For example, consider the *radix character*, the symbol used to separate the integer portion of a number from the fractional portion. In American English, this character is a period; in much of Europe, it is a comma. Conversely, the thousands separator that separates numbers larger than three digits is a comma in American English, and a period in much of Europe.

The convention for grouping digits also varies. In American English, digits are grouped by threes, but there are many other possibilities. In the example below, the same number is written as it would be locally in three different countries:

1,000,000.55	US
1.000.000,55	Germany
10,00,000.55	Nepal

1.2.1.3 Currency

We are all aware that countries use different currencies. However, not everyone realizes the many different ways we can represent units of currency. For example, the symbol for a currency can vary. Here are two different ways of representing the same amount in US dollars:

$24.99	US
USD 24.99	International currency symbol for the US

The placement of the currency symbol varies for different currencies, too, appearing before, after, or even within the numeric value:

¥ 155	Japan
13,50 DM	Germany
£14 19s. 6d.	England before decimalization

The format of negative currency values differs:

öS 1,1	-öS 1,1	Austria
1,1 DM	-1,1 DM	Germany
SFr. 1.1	SFr.-1.1	Switzerland
HK$1.1	(HK$1.1)	Hong Kong

1.2.1.4 Time and Date

Local conventions also determine how *time* and *date* are displayed. Some countries use a 24-hour clock; others use a 12-hour clock. Names and abbreviations for days of the week and months of the year vary by language.

Customs dictate the ordering of the year, month, and day, as well as the separating delimiters for their numeric representation. To designate years, some regions use seasonal, astronomical, or historical criteria, instead of the Western Gregorian calendar system. For example, the official Japanese calendar is based on the year of reign of the current Emperor.

The following example shows short and long representations of the same date in different countries:

10/29/96	Tuesday, October 29, 1996	US
1996. 10. 29.	1996. október 29.	Hungary
29/10/96	martedì 29 ottobre 1996	Italy
29/10/1996	Τριτη, 29 Οκτωβριου 1996	Greece
29.10.96	Dienstag, 29. Oktober 1996	Germany

The following example shows different representations of the same time:

4:55 pm	US time
16:55 Uhr	German time

And the following example shows different representations of the same time:

11:45:15	Digital representation, US
11:45:15 µµ	Digital representation, Greece

1.2.1.5 Ordering

Languages may vary regarding *collating sequence* ; that is, their rules for ordering or sorting characters or strings. The following example shows the same list of words ordered alphabetically by different collating sequences:

Sorted by [1]ASCII rules	Sorted by German rules
Airplane	Airplane
Zebra	ähnlich
bird	bird
car	car
ähnlich	Zebra

The ASCII collation orders elements according to the numeric value of bytes, which does not meet the requirements of English language dictionary sorting. This is because lexicographical order sorts `a` after `A` and before `B`, whereas ASCII-based order sorts `a` after the entire set of uppercase letters.

The German alphabet sorts `ä` before `b`, whereas the ASCII order sorts an umlaut after all other letters.

In addition to specifying the ordering of individual characters, some languages specify that certain groups of characters should be

[1] ASCII stands for American Standard Code for Information Interchange. A 7-bit code is used in the US.

clustered and treated as a single character. The following example shows the difference this can make in an ordering:

Sorted by ASCII rules	Sorted by Spanish rules
chaleco	cuna
cuna	chaleco
día	día
llava	loro
loro	llava
maíz	maíz

The word `llava` is sorted after `loro` and before `maíz`, because in Spanish `ll` is a digraph[2], i.e., it is treated as a single character that is sorted after `l` and before `m`. Similarly, the digraph `ch` in Spanish is treated as a single character to be sorted after `c`, but before `d`. Two characters that are paired and treated as a single character are referred to as a *two-to-one character code pair*.

In other cases, one character is treated as if it were actually two characters. The German single character `ß`, called the *sharp s*, is treated as `ss`. This treatment makes a difference in the ordering, as shown in the example below:

Sorted by ASCII rules	Sorted by German rules
Rosselenker	Rosselenker
Rostbratwurst	Roßhaar
Roßhaar	Rostbratwurst

1.2.2 Character Encodings for Localizing Alphabets

We know that different languages can have different alphabets. The first step in localizing an alphabet is to find a way to represent,

[2] Generally, a digraph is a combination of characters that is written separately, but forms a single lexical unit.

or *encode* , all its characters. In general, alphabets may have different *character encodings* .

The *7-bit ASCII codeset* is the traditional code on UNIX systems.

The *8-bit codesets* permit the processing of many Eastern and Western European, Middle Eastern, and Asian Languages. Some are strictly extensions of the 7-bit ASCII codeset; these include the 7-bit ASCII codes and additionally support 128-character codes beyond those of ASCII. Such extensions meet the needs of Western European users. To support languages that have completely different alphabets, such as Arabic and Greek, larger 8-bit codesets have been designed.

Multibyte character codes are required for alphabets of more than 256 characters, such as kanji, which consists of Japanese ideographs based on Chinese characters. Kanji has tens of thousands of characters, each of which is represented by two bytes. To ensure backward compatibility with ASCII, a multibyte codeset is a superset of the ASCII codeset and consists of a mixture of one- and two-byte characters.

For such languages, several encoding schemes have been defined. These encoding schemes provide a set of rules for parsing a byte stream into a group of coded characters.

1.2.2.1 Multibyte Encodings

Handling multibyte character encodings is a challenging task. It involves parsing multibyte character sequences, and in many cases requires conversions between multibyte characters and wide characters.

Understanding multibyte encoding schemes is easier when explained by means of a typical example. One of the earliest and probably biggest markets for multibyte character support is in Japan. Therefore, the following examples are based on encoding schemes for Japanese text processing.

In Japan, a single text message can be composed of characters from four different writing systems. *Kanji* has tens of thousands of characters, which are represented by pictures. *Hiragana* and *katakana* are syllabaries, each containing about 80 sounds, which are also represented as ideographs. The *Roman* characters include some 95 letters, digits, and punctuation marks.

Here is an example of an encoded Japanese sentence composed of these four writing systems:

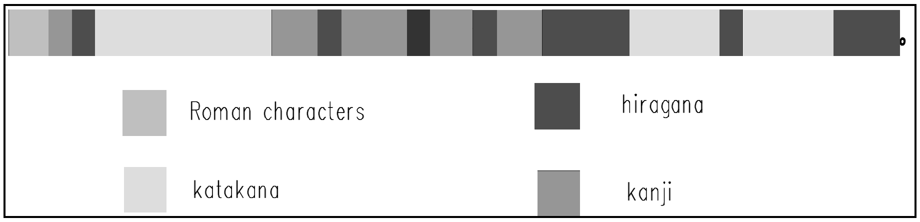


Figure 1. A Japanese sentence mixing four writing systems

The sentence means: "Encoding methods such as JIS can support texts that mix Japanese and English."

A number of Japanese character sets are common:

JIS C 6226-1978	JIS X 0208-1983
JIS X 0208-1990	JIS X 0212-1990
JIS-ROMAN	ASCII

There is no universally recognized multibyte encoding scheme for Japanese. Instead, we deal with the three common multibyte encoding schemes defined below:

JIS (Japanese Industrial Standard)

Shift-JIS

EUC (Extended UNIX Code)

1.2.2.1.1 JIS Encoding

The *JIS*, or *Japanese Industrial Standard*, supports a number of standard Japanese character sets, some requiring one byte, others two. Escape sequences are required to shift between one- and two-byte modes.

Escape sequences, also referred to as *shift sequences*, are sequences of *control characters*. Control characters do not belong to any of the alphabets. They are artificial characters that do not have a visual representation. However, they are part of the encoding scheme, where they serve as separators between different character sets, and indicate a switch in the way a character sequence is interpreted. The use of the shift sequence is demonstrated in the following figure.

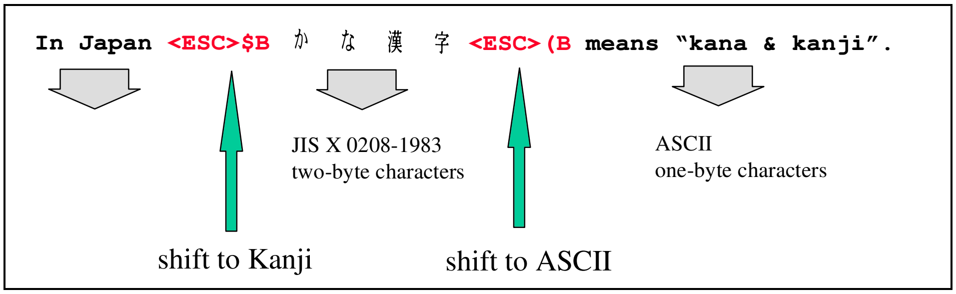


Figure 2. An example of a Japanese text encoded in JIS

For encoding schemes containing shift sequences, like JIS, it is necessary to maintain a *shift state*while parsing a character sequence. In the example above, we are in some initial shift state at the start of the sequence. Here it is ASCII. Therefore, characters are assumed to be one-byte ASCII codes until the shift sequence`<ESC>$B` is seen. This switches us to two-byte mode, as defined by JIS X 0208-1983. The shift sequence`<ESC>(B` then switches us back to ASCII mode.

Encoding schemes that use shift state are not very efficient for internal storage or processing. Sometimes shift sequences require up to six bytes. Frequent switching between character sets in a file of strings could cause the number of bytes used in shift sequences to exceed the number of bytes used to represent the actual data!

Encodings containing shift sequences are used primarily as an external code, which allows information interchange between a program and the outside world.

1.2.2.1.2 Shift-JIS Encoding

Despite its name, Shift-JIS has nothing to do with shift sequences and states. In this encoding scheme, each byte is inspected to see if it is a one-byte character or the first byte of a two-byte character. This is determined by reserving a set of byte values for certain purposes. For example:

1. Any byte having a value in the range 0x21-7E is assumed to be a one-byte ASCII/JIS Roman character.
2. Any byte having a value in the range 0xA1-DF is assumed to be a one-byte half-width katakana character.
3. Any byte having a value in the range 0x81-9F or 0xE0-EF is assumed to be the first byte of a two-byte character from the set JIS X 0208-1990. The second byte must have a value in the range 0x40-7E or 0x80-FC.

While this encoding is more compact than JIS, it cannot represent as many characters as JIS. In fact, Shift-JIS cannot represent any characters in the supplemental character set JIS X 0212-1990, which contains more than 6,000 characters.

1.2.2.1.3 EUC Encoding

EUC is not peculiar to Japanese encoding. It was developed as a method for handling multiple character sets, Japanese or otherwise, within a single text stream.

The EUC encoding is much more extensible than Shift-JIS since it allows for characters containing more than two bytes. The encoding scheme used for Japanese characters is as follows:

1. Any byte having a value in the range 0x21-7E is assumed to be a one-byte ASCII/JIS Roman character.
2. Any byte having a value in the range 0xA1-FE is assumed to be the first byte of a two-byte character from the set JIS X0208-1990. The second byte must also have a value in that range.
3. Any byte having a value in the range 0x8E is assumed to be followed by a second byte with a value in the range 0xA1-DF, which represents a half-width katakana character.
4. Any byte having the value 0x8F is assumed to be followed by two more bytes with values in the range 0xA1-FE, which together represent a character from the set JIS X0212-1990.

The last two cases involve a prefix byte with values 0x8E and 0x8F, respectively. These bytes are somewhat like shift sequences in that they introduce a change in subsequent byte interpretation. However, unlike the shift sequences in JIS, which introduce a sequence, these prefix bytes must precede *every* multibyte character, not just the first in a sequence. For this reason, each multibyte character encoded in this manner stands alone and EUC is not considered to involve shift states.

1.2.2.1.4 Uses of the Three Multibyte Encodings

The three multibyte encodings just described are typically used in separate areas:

- *JIS* is the primary encoding method used for electronic transmission such as email because it uses only 7 bits of each byte. This is required because some network paths strip the eighth bit from characters. Escape sequences are used to switch between one- and two-byte modes, as well as between different character sets.

- *Shift-JIS* was invented by Microsoft and is used on MS-DOS-based machines. Each byte is inspected to see if it is a one-byte character or the first byte of a two-byte character. Shift-JIS does not support as many characters as JIS and EUC do.
- *EUC* encoding is implemented as the internal code for most UNIX-based platforms. It allows for characters containing more than two bytes, and is much more extensible that Shift-JIS. EUC is a general method for handling multiple character sets. It is not peculiar to Japanese encoding.

1.2.2.2 Wide Characters

Multibyte encoding provides an efficient way to move characters around outside programs, and between programs and the outside world. Once inside a program, however, it is easier and more efficient to deal with characters that have the same size and format. We call these *wide characters*.

An example will illustrate how wide characters make text processing inside a program easier. Consider a filename string containing a directory path where adjacent names are separated by a slash—for example, `/CC/include/locale.h`. To find the actual filename in a single-byte character string, we can start at the back of the string. When we find the first separator, we know where the filename starts. If the string contains multibyte characters, we must scan from the front so we don't inspect bytes out of context. If the string contains wide characters, however, we can treat it like a single-byte character and scan from the back.

Conceptually, you can think of wide character sets as being extended ASCII or EBCDIC[3]; each unique character is assigned a distinct value. Since they are used as the counterpart to a multibyte encoding, wide character sets must allow representation of all characters that can be represented in a multibyte encoding as wide characters. As multibyte encodings support thousands of characters, wide characters are usually larger that one byte—typically two or four bytes. All characters in a wide character set are of equal size. The size of a wide character is not universally fixed, although this depends on the particular wide character set.

There are a number of wide character standards, including those shown below:

---

[3] EBCDIC stands for "extended binary coded decimal interchange code. It is a single-byte character set developed by IBM.

ISO 10646.UCS-2[4]	16-bit characters
ISO 10646.UCS-4	32-bit characters
Unicode[5]	16-bit characters

The programming language C++ supports wide characters; their native type in C++ is called `wchar_t`. The syntax for wide character constants and wide character strings is similar to that for ordinary, tiny character constants and strings:

`L'a'` is a wide character constant, and

`L"abc"` is a wide character string.

1.2.2.3 Conversion between Multibytes and Wide Characters

Since wide characters are usually used for internal representation of characters in a program, and multibyte encodings are used for external representation, converting multibytes to wide characters is a common task during input/output operations. Input to and output from files is a typical example. The file will usually contain multibyte characters. When you read such a file, you convert these multibyte characters into wide characters that you store in an internal wide character buffer for further processing. When you write to a multibyte file, you have to convert the wide characters held internally into multibytes for storage on a external file. The following figure demonstrates graphically how this conversion during file input is done:

---

[4] ISO 10646 is the encoding of the International Standards Organization.

[5] Unicode was developed by the Unicode Consortium. It is code-for-code equivalent to the 16-bit ISO 10646 encoding.

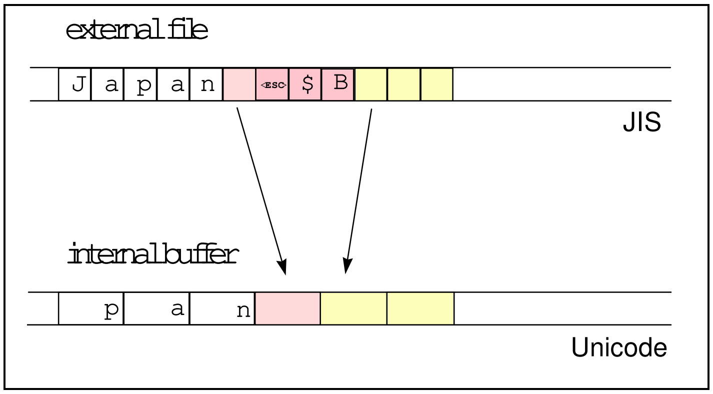


Figure 3. Conversion from a multibyte to a wide character encoding

The conversion from a multibyte sequence into a wide character sequence requires expansion of one-byte characters into two- or four-byte wide characters. Escape sequences are eliminated. Multibytes that consist of two or more bytes are translated into their wide character equivalents.

1.2.3 Summary

In this section, we discussed a variety of issues involved in developing software for worldwide use. For all of these areas in which cultural conventions differ from one region to another, the *Standard C++ Library* provides services that enable you to easily internationalize your C++ programs. These services include:

- Formatting and parsing of numbers, currency unit, dates, and time;
- Handling different alphabets, their character classification, and collation sequences;
- Converting codesets, including multibyte to wide character conversion;
- Handling messages in different languages.

1.3 The Standard C Locale and the Standard C++ Locales

As a software developer, you may already have some background in the C programming language, and the internationalization services provided by the C library. You may even be facing the problem of integrating internationalized software written in C with software in C++. If so, we recommend that you study this section. Here we give a short recap of the internationalization services provided by the C library, and its relationship to C++ locales. We then describe the C++ locales in terms of the C locale.

1.3.1 The C Locale

All the culture and language dependencies discussed in the previous section need to be represented in an operating system. This information is usually represented in a kind of language table, called a *locale*.

The X/Open consortium has standardized a variety of services for *Native Language Support (NLS)* in the programming language C. This standard is commonly known as XPG4. The X/Open's *Native Language Support* includes internationalization services as well as localization support.[6] The description below is based on this standard.

According to XPG4, the C locale is composed of several *categories*:

Table 1. Categories of the C locale

Category	Content
`LC_NUMERIC`	Rules and symbols for numbers
`LC_TIME`	Values for date and time information
`LC_MONETARY`	Rules and symbols for monetary information
`LC_CTYPE`	Character classification and case conversion
`LC_COLLATE`	Collation sequence
`LC_MESSAGE`	Formats and values of messages

[6] ISO C also defines internationalization services in the programming language C. The respective ISO standard is ISO/IEC 9899 and its Amendment 1. The ISO C standard is identical to the POSIX standard for the programming language C. The internationalization services defined by ISO C are part of XPG4. However, XPG4 offers more services than ISO C, such as localization support.

The external representation of a C locale is usually as a *file* in UNIX. Other operating systems may choose other representations. The external representation is transformed into an internal memory representation by calling the function `setlocale()`, as shown in the figure below:

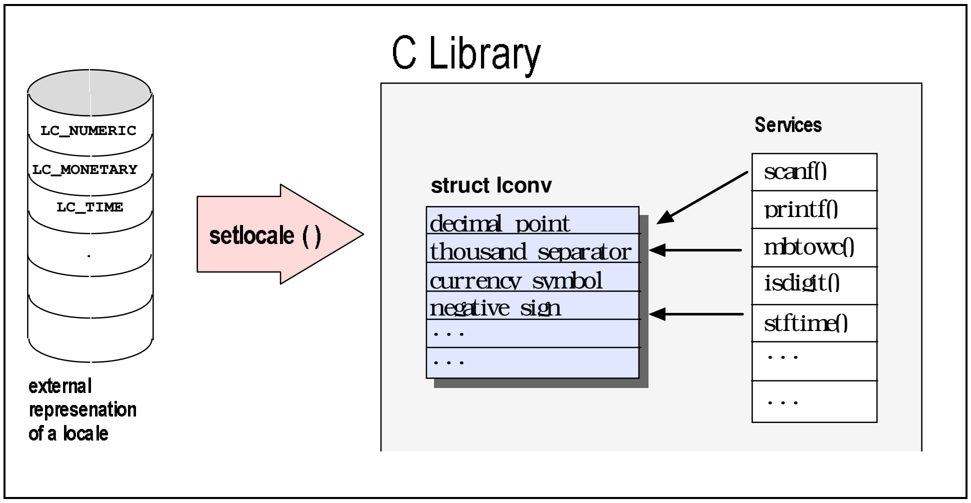


Figure 4. Transformation of a C locale from external to internal representation

Inside a program, the C locale is represented by one or more global *data structures*. The C library provides a *set of functions* that use information from those global data structures to adapt their behavior to local conventions. Examples of these functions and the information they cover are listed below:

Table 2. C locale functions and the information they cover

C locale function	Information covered
`setlocale()`, ...	Locale initialization and language information
`isalpha()`, `isupper()`, `isdigit()`, ...	Character classification
`strftime()`, ...	Date and time functions
`strfmon()`	Monetary functions
`printf()`, `scanf()`, ...	Number parsing and formatting
`strcoll()`, `wcscoll()`, ...	String collation

`mblen()`, `mbtowc()`, `wctomb()`, ...	Multibyte functions
`cat_open()`, `catgets()`, `cat_close()`	Message retrieval

1.3.2 The C++ Locales

In C++, a locale is a class called `locale` provided by the Standard C++ Library. The C++ class `locale` differs from the C locale because it is more than a language table, or data representation of the various culture and language dependencies. It also includes the internationalization services, which in C are global functions.

In C++, internationalization semantics are broken out into separate classes called *facets*. Each facet handles a set of internationalization services, for example, the formatting of monetary values. Facets may also represent a set of culture and language dependencies, such as the rules and symbols for monetary information.

Each locale object maintains a set of facet objects. Basically, you can think of a C++ locale as a container of facets. This concept is illustrated graphically in the figure below:

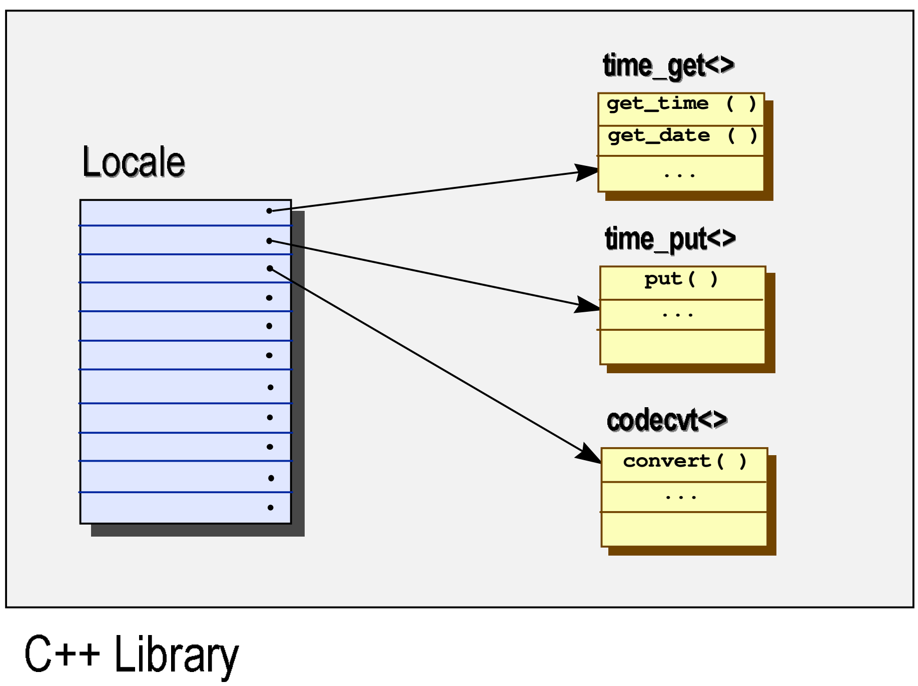


Figure 5. A C++ locale is a container of facets

1.3.3 Facets

Facet classes encapsulate data that represents a set of culture and language dependencies, and offer a set of related internationalization services. Facet classes are very flexible. They can contain just about any internationalization service you can invent. The *Standard C++ Library* offers a number of predefined *standard facets*, which provide services similar to those contained in the C library. However, you are free to bundle additional internationalization services into a new facet class, or purchase a library of facets.

1.3.3.1 The Standard Facets

As listed in Table 1, the C locale is composed of six categories of locale-dependent information: `LC_NUMERIC` (rules and symbols for numbers), `LC_TIME` (values for date and time information), `LC_MONETARY` (rules and symbols for monetary information), `LC_CTYPE` (character classification and conversion), `LC_COLLATE` (collation sequence), and `LC_MESSAGE` (formats and values of messages).

Similarly, there are six groups of standard facet classes. A detailed description of these facets is contained in the *Class Reference*, but a brief overview is given below. Note that an abbreviation like `num_get <charT,InputIterator>` means that `num_get` is a class template taking two template arguments, a character type, and an input iterator type. The groups of the standard facets are:

- Numeric. The facet classes `num_get<charT,InputIterator>` and `num_put<charT, OutputIterator>` handle numeric formatting and parsing. The facet classes provide `get()` and `put()` member functions for values of type `long`, `double`, etc.

  The facet class `numpunct<charT>` specifies numeric punctuation. It provides functions like `decimal_point()`, `thousands_sep()`, etc.

- Monetary. The facet classes `money_get<charT,bool,InputIterator>` and `money_put<charT, bool, OutputIterator>` handle formatting and parsing of monetary values. They provide `get()` and `put()` member functions that parse or produce a sequence of digits, representing a count of the smallest unit of the currency. For example, the sequence $1,056.23 in a common US locale would yield 105623 units, or the character sequence "105623".

  The facet class `moneypunct <charT, bool International>` handles monetary punctuation like the facet `numpunct<charT>` handles numeric punctuation. It comes with functions like `curr_symbol()`, etc.

- Time. The facet classes `time_get<charT,InputIterator>` and `time_put<charT, OutputIterator>` handle date and time formatting and parsing. They provide functions like `get_time()`, `get_date()`, `get_weekday()`,etc.
- Ctype. The facet class `ctype<charT>` encapsulates the Standard C++ Library `ctype` features for character classification, like `tolower()`, `toupper()`, `isspace()`, `isprint()`, etc.
- Collate. The facet class `collate<charT>` provides features for string collation, including a `compare()` function used for string comparison.
- Code Conversion. The facet class `codecvt<fromT,toT,stateT>` is used when converting from one encoding scheme to another, such as from the multibyte encoding JIS to the wide-character encoding Unicode. Instances of this facet are typically used in pairs. The main member function is `convert()`. There are template specializations `<char, wchar_t, mbstate_t>` and `<wchar_t, char, mbstate_t>` for multibyte to wide character conversions.
- Messages. The facet class `messages<charT>` implements the X/Open message retrieval. It provides facilities to access message catalogues via `open()` and `close(catalog)`, and to retrieve messages via `get(..., int msgid,...)`.

The names of the standard facets obey certain naming rules. The `get` facet classes, like `num_get` and `time_get`, handle parsing. The `put` facet classes handle formatting. The `punct` facet classes, like `numpunct` and `moneypunct`, represent rules and symbols.

1.3.4 Differences between the C Locale and the C++ Locales

As we have seen so far, the C locale and the C++ locale offer similar services. However, the semantics of the C++ locale are different from the semantics of the C locale:

- The *Standard C locale* is a global resource: there is only one locale for the entire application. This makes it hard to build an application that has to handle several locales at a time.
- The *Standard C++ locale* is a class. Numerous instances of class `locale` can be created at will, so you can have as many locale objects as you need.

To explore this difference in further detail, let us see how locales are typically used.

1.3.4.1 Common Uses of the C locale

The C locale is commonly used as a default locale, a native locale, or in multiple locale applications.

Default locale. As a developer, you may never require internationalization features, and thus never set a locale. If you can safely assume that users of your applications are accommodated by the classic US English ASCII behavior, you have no need for localization. Without even knowing it, you will always use the default locale, which is the US English ASCII locale.

Native locale. If you do plan on localizing your program, the appropriate strategy may be to retrieve the native locale once at the beginning of your program, and never, ever change this setting again. This way your application will adapt itself to one particular locale, and use this throughout its entire run time. Users of such applications can explicitly set their favorite locale before starting the application. Usually the system's default settings will automatically activate the native locale.

Multiple locales. It may well happen that you do have to work with multiple locales. For example, if you have to implement an application for Switzerland, you might want to output messages in Italian, French, and German. As the C locale is a global data structure, you will have to switch locales several times.

Let's look at an example of an application that works with multiple locales. Imagine an application that prints invoices to be sent to customers all over the world. Of course, the invoices need to be printed in the customer's native language, so the application has to write output in multiple languages. Prices to be included in the invoice are taken from a single price list. If we assume the application is used by a US company, the price list will be in US English.

The application reads input (the product price list) in US English, and writes output (the invoice) in the customer's native language, say German. Since there is only one global locale in C that affects both input and output, the global locale must change between input and output operations. Before a price is read from the English price list, the locale must be switched from the German locale used for printing the invoice to a US English locale. Before inserting the price into the invoice, the global locale must be switched back to the German locale. To read the next input from the price list, the locale must be switched back to English, and so forth. This activity is summarized in the following figure:

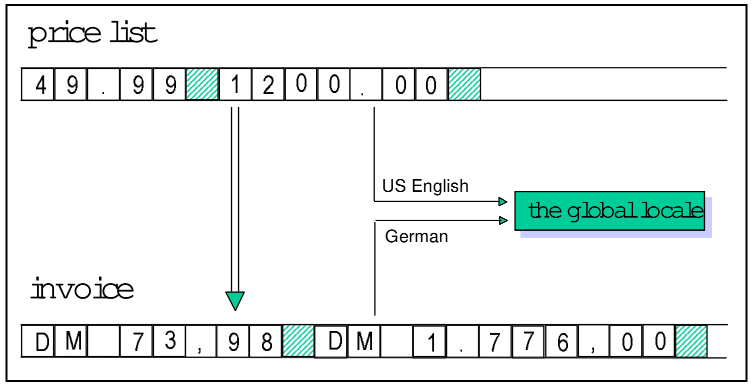


Figure 6. Multiple locales in C

Here is the C code that corresponds to the previous example[7]:

```
double price;
char buf[SZ];
while ( … )  // processing the German invoice
{  setlocale(LC_ALL, "En_US");
   fscanf(priceFile,"%fl",&price);
   // convert $ to DM according to the current exchange rate
   setlocale(LC_ALL,"De_DE");
   fprintf(invoiceFile,"%f",price);
}
```

Using C++ locale objects dramatically simplifies the task of communicating between multiple locales. The iostreams in the *Standard C++ Library* are internationalized so that streams can be *imbued* with separate locale objects. For example, the input stream can be imbued with an English locale object, and the output stream can be imbued with a German locale object. In this way, switching locales becomes unnecessary, as demonstrated in the figure below:

---

[7] The example is oversimplified. One would certainly use the `strfmon()` function for formatting monetary values like prices. We will consider more realistic examples in section 1.5.

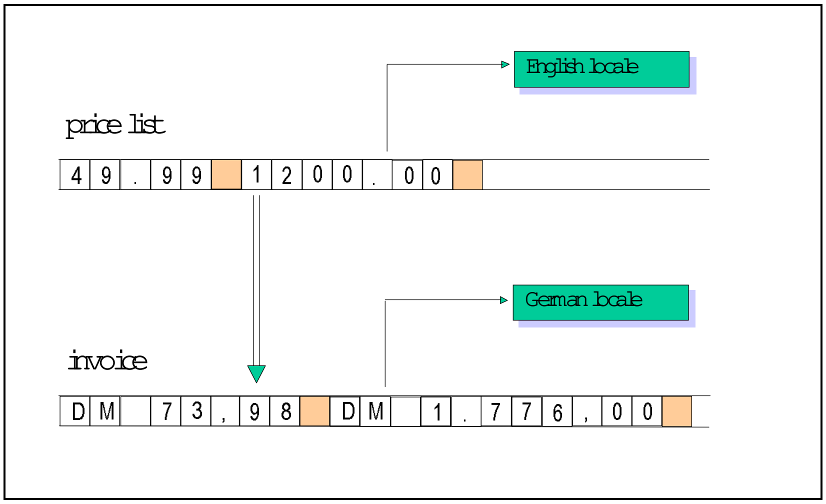


Figure 7. Multiple locales in C++

Here is the C++ code corresponding to the previous example:

```
priceFile.imbue(locale("En_US"));
invoiceFile.imbue(locale("De_DE");
double price;
while ( … )  // processing the German invoice
{  priceFile >> price;
   // convert $ to DM according to the current exchange rate
   invoiceFile << price;
}
```

Because the examples given above are brief, switching locales might look like a minor inconvenience. However, it is a major problem once code conversions are involved.

To underscore the point, let us revisit the JIS encoding scheme using the shift sequence described in Figure 2, and repeated below. With these encodings, you will recall that you must maintain a *shift state* while parsing a character sequence, as shown in Figure 8:

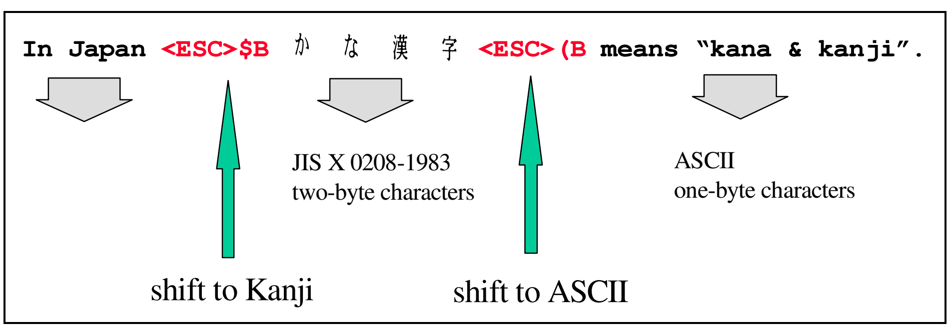


Figure 8. The Japanese text encoded in JIS from Figure 2

Suppose you are parsing input from a multibyte file that contains text that is encoded in JIS, as shown in the next figure. While you parse this file, you have to keep track of the current shift state so you know how to interpret the characters you read, and how to transform them into the appropriate internal wide character representation.

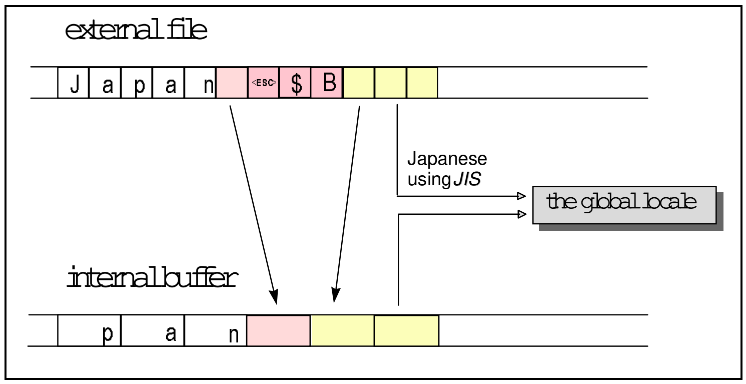


Figure 9. Parsing input from a multibyte file using the global C locale

The global C locale can be switched during parsing; for example, from a locale object specifying the input to be in JIS encoding, to a locale object using EUC encoding instead. The current shift state becomes invalid each time the locale is switched, and you have to carefully maintain the shift state in an application that switches locales.

As long as the locale switches are intentional, this problem can presumably be solved. However, in multithreaded environments, the global C locale may impose a severe problem, as it can be switched inadvertently by another otherwise unrelated thread of execution. For this reason, internationalizing a C program for a multithreaded environment is difficult.

If you use C++ locales, on the other hand, the problem simply goes away. You can imbue each stream with a separate locale object, making inadvertent switches impossible.

Let us now see how C++ locales are intended to be used.

1.3.4.2 Common Uses of C++Locales

The C++ locale is commonly used as a default locale, with multiple locales, and as a global locale.

Default locale. If you are not involved with internationalizing programs, you won't need C++ locales any more than you need C locales. If you can safely assume that users of your applications are accommodated by classic US English ASCII behavior, you will not require localization features. For you, the *Standard C++ Library* provides a predefined locale object, `locale::classic()`, that represents the US English ASCII locale.

Multiple locales. Working with many different locales becomes easy when you use C++ locales. Switching locales, as you did in C, is no longer necessary in C++. You can imbue each stream with a different locale object. You can pass locale objects around and use them in multiple places.

Global locale. There is a global locale in C++, as there is in C. You can make a given locale object global by calling `locale::global()`. You can create snapshots of the current global locale by calling the default constructor for a locale `locale::locale()`. Snapshots are immutable locale objects and are not affected by any subsequent changes to the global locale. Internationalized components like iostreams use it as a default. If you do not explicitly imbue your streams with any particular locale object, a snapshot of the global locale is used.

Using the global C++ locale, you can work much as you did in C. You activate the native locale once at program start — in other words, you make it global — and use snapshots of it thereafter for all tasks that are locale-dependent. The following code demonstrates this procedure:

```
locale::global(locale(""));                              //1
…
string t = print_date(today, locale());                  //2
…
locale::global(locale("Fr_CH"));                         //3
…
cout << something;                                       //4
```

`//1` Make the native locale global.

`//2` Use snapshots of the global locale whenever you need a locale object. Assume that `print_date()` is a function that formats dates. You would provide the function with a snapshot of the global locale in order to do the formatting.

`//3` Switch the global locale; make a French locale global.

`//4` Note that you need not explicitly imbue any streams with the global locale. They use a snapshot of the global locale by default.

1.3.5 Relationship between the C Locale and the C++ Locale

The C locale and the C++ locales are mostly unrelated. However, making a C++ locale object global via `locale::global()` affects the global C locale and results in a call to `setlocale()`. When this happens, locale-sensitive C functions called from within a C++ program will use the global C++ locale.

There is no way to affect the C++ locale from within a C program.

1.4 The Locale

A C++ locale object is a container of facet objects that encapsulates internationalization services and represent culture and language dependencies. Here are some functions of class `locale` which allow you to create locales:

```
class locale {
    public:
// construct/copy/destroy:
      explicit locale(const char* std_name);                  \\1
// global locale objects:
      static const locale& classic();                         \\2
};
```

//1 You can create a locale object from a C locale's external representation. The constructor `locale::locale(const char* std_name)` takes the name of a C locale. This locale name is like the one you would use for a call to the C library function `setlocale()`.

//2 You can also use a predefined locale object, `locale :: classic()`, which represents the US English ASCII environment.

For a comprehensive description of the constructors described above, see the *Class Reference*.

It's important to understand that locales are immutable objects: once a locale object is created, it cannot be modified. This makes locales reliable and easy to use. As a programmer, you know that whenever you use pointers or references to elements held in a container, you have to worry about the validity of the pointers and references. If the container changes, pointers and references to its elements might not be valid any longer.

A locale object is a container, too. However, it is an immutable container; that is, it does not change. Therefore, you can take a reference to a locale's facet object and pass the reference around without worrying about the validity of this reference. The related locale object will never be modified; no facets can be silently replaced.

At some time, you will most likely need locale objects other than the US classic locale or a snapshot of the global locale. Since locales are immutable objects, however, you cannot take one of these and replace its facet objects. You have to say at construction time how they shall be built.

Here are some constructors of class `locale` which allow you to build a locale object by composition; in other words, you construct it by copying an existing locale object, and replacing one or several facet objects.

```
class locale {
public:
  locale(const locale& other, const char* std_name, category);
  template <class Facet> locale(const locale& other, Facet* f);
  template <class Facet> locale(const locale& other
                               ,const locale& one);
  locale(const locale& other, const locale& one, category);
};
```

The following example shows how you can construct a locale object as a copy of the classic locale object, and take the numeric facet objects from a German locale object:

```
locale loc ( locale::classic(), locale("De_DE"), LC_NUMERIC );
```

For a comprehensive description of the constructors described above, see the *Class Reference*.

Copying a locale object is a cheap operation. You should have no hesitation about passing locale objects around by value. You may copy locale objects for composing new locale objects; you may pass copies of locale objects as arguments to functions, etc.

Locales are implemented using reference counting and the handle-body-idiom[8]: When a locale object is copied, only its handle is duplicated—a fast and inexpensive action. Similarly, constructing a locale object with the default constructor is cheap— this is equivalent to copying the global locale object. All other locale constructors that take a second locale as an argument are moderately more expensive, because they require cloning the body of the locale object. However, the facets are not all copied. The byname constructor is the most expensive, because it requires creating the locale from an external locale representation.

[8] A good reference for an explanation of the handle-body idiom is: "Advanced C++ Programming St yles and Idioms," James O. Coplien, Addison-Wesley, 1992, ISBN 0-201-54855-0.

The following figure describes an overview of the locale architecture. It is a handle to a body that maintains a vector of pointers of facets. The facets are reference-counted, too.

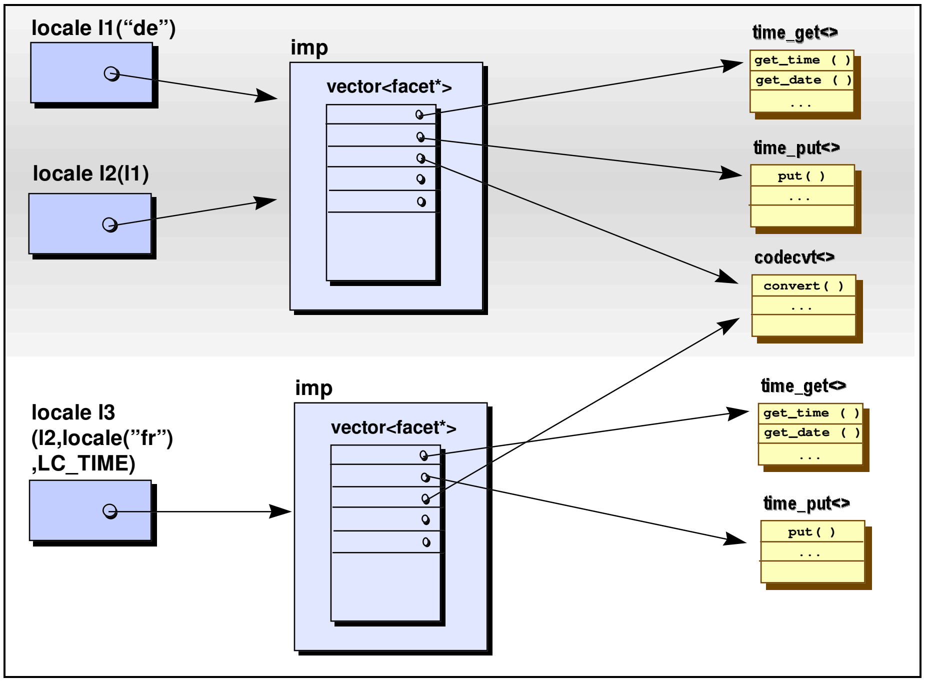


Figure 10. The locale architecture

1.5 The Facets

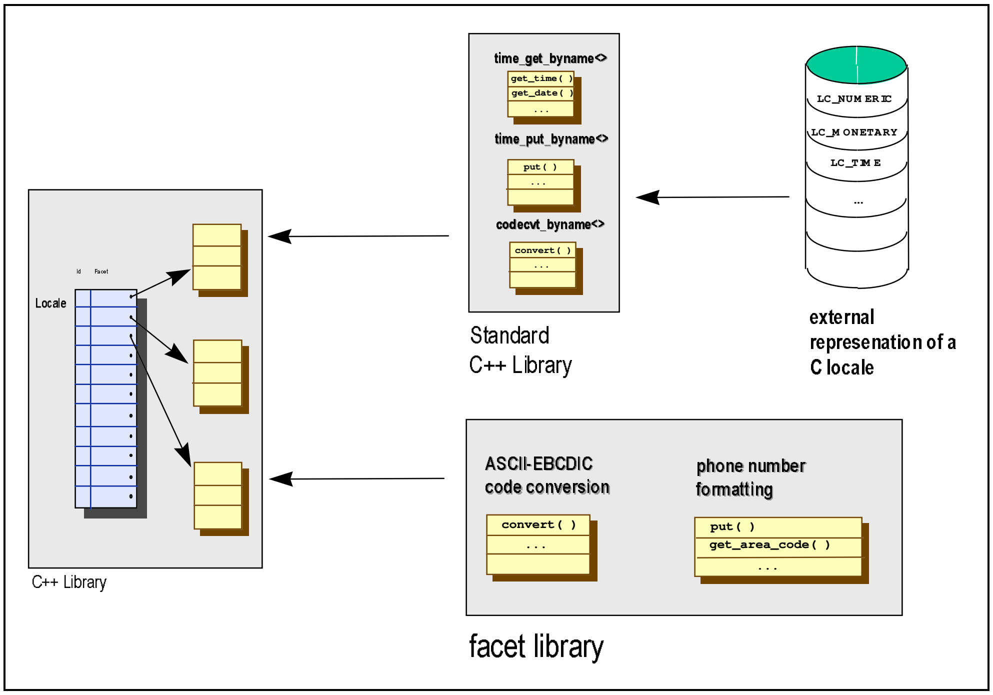
A facet is a nested class inside class `locale`; it is called `locale::facet`. Facet objects encapsulate internationalization services, and represent culture and language dependencies.

1.5.1 Creating a Facet Object

There are several ways to create facet objects:

- Buy a facet library, which provides you with facet classes and objects.
- Build your own facet classes and construct facet objects.

- Build facet objects from the external representation of a C locale. This is done via the constructor of one of the byname[9] facet classes from the Standard C++ Library, as shown in the figure



below:

Figure 11. Creating facet objects

Facets are interdependent. For example, the `num_get` and `num_put` facet objects rely on a `numpunct` facet object. In most cases, facet objects will not be used independently of each other, but will be grouped together in a locale object. For this reason, facet objects are usually constructed along with the locale object that maintains them. You will only rarely need to construct a single facet for stand-alone use.

However, the example below demonstrates how you would write the code to construct and use a single facet object if needed. It is an

[9] A byname facet creates a facet from the external representation of a C locale. See section 1.3.3.1 for the naming conventions of facet names.

example of a locale-sensitive string comparison, which in C you would perform using the `strcoll()` function.[10]

```
string name1("Peter Gartner");
string name2 ("Peter Gärtner");
collate_byname<char> collFacet("De_DE");                      \\1
if ( collFacet.compare                                         \\2
    (name1.begin(), name1.end(), name2.begin(), name2.end())
    == -1)
{ … }
```

//1 A collation facet object is constructed. It is created from a German C locale's external representation.

//2 The member function `compare()` of this facet object is used for string comparison.

1.5.2 Accessing a Locale's Facets

A locale object is like a container— or a map, to be more precise— but it is indexed by type at compile time. The indexing operator, therefore, is not`operator[]`, but rather the template operator `<>`. Access to the facet objects of a locale object is via two member function templates, `use_facet` and `has_facet`:

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

The code below demonstrates how they are used. It is an example of the `ctype` facet's usage; all upper case letters of a string read from the standard input stream are converted to lower case letters and written to the standard output stream.

```
string in;
cin >> in;
if (has_facet< ctype<char> >(locale::locale()))               \\1
{  cout << use_facet< ctype<char> >(locale::locale())         \\2
               .tolower(in.begin(),in.end());                 \\3
}
```

//1 In the call to `has_facet<...>()`, the template argument chooses a facet class. If no object of the named facet class is present in a locale object, `has_facet` returns`false`.

//2 The function template `use_facet<…>()` returns a reference to a locale's facet object. As locale objects are immutable, the

[10] The string class in the Standard C++ Library does not provide any service for locale-sensitive string comparisons. Hence, you will usually use a collate facet's compare service instead.

reference to a facet object obtained via `use_facet()` stays valid throughout the lifetime of the locale object.

//3 The facet object's member function `tolower()` is called. It has the functionality of the C function `tolower()`; it converts all upper case letters into lower case letters.

In most situations, you do not have to check whether a locale has a standard facet object like `ctype`. Most locale objects are created by composition, starting with a locale object constructed from a C locale's external representation. Locale objects created this way, that is, via a byname constructor, always have all of the standard facet objects. Because you can only add or replace facet objects in a locale object, you cannot compose a locale that misses one of the standard facets.

A call to `has_facet()` is useful though, when you expect that a certain non-standard facet object should be present in a locale object.

1.5.3 Using a Stream's Facet

Here is a more advanced example that uses a time facet for printing a date. Let us assume we have a date and want to print it this way:

```
struct tm aDate;                                         //1
aDate.tm_year = 1989;
aDate.tm_mon = 9;
aDate.tm_mday = 1;                                       //2

cout.imbue(locale::locale("De_CH"));                     //3
cout << aDate;                                           //4
```

//1 A date object is created. It is of type `tm`, which is the time structure defined in the standard C library.

//2 The date object is initialized with a particular date, September 1, 1989.

//3 Let's assume our program is supposed to run in a German-speaking canton of Switzerland. Hence, a Swiss locale is attached to the standard output stream.

//4 The date is printed in German to the standard output stream.

The output will be: `1. September 1989`

As there is no `operator<<()` defined in the Standard C++ Library for the time structure `tm` from the C library, we have to provide this inserter ourselves. The following code suggests a way this can be done. If you are not familiar with iostreams, you will want to refer to the iostreams section of this User's *Guide* for more information.

To keep it simple, the handling of exceptions thrown during the formatting is omitted.

```
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT,traits>& os, const tm& date)   \\1
{
 locale loc = os.getloc();                                     \\2
 typedef ostreambuf_iterator<charT,traits> outIter_t;          \\3
 const time_put<charT,outIter_t>& fac;                         \\4
 fac = use_facet < time_put<charT, bufIter_t > > (loc);        \\5
 outIter_t nxtpos;                                             \\6
 nxtpos = fac.put(os,os,os.fill(),&date,'x');                  \\7
 if (nxtpos.failed())                                          \\8
    os.setstate(ios_base::badbit);                             \\9
 return os;
}
```

//1 This is a typical signature of a stream inserter; it takes a reference to an output stream and a constant reference to the object to be printed, and returns a reference to the same stream.

//2 The stream's locale object is obtained via the stream's member function `getloc()`. This is the locale object where we expect to find a time-formatting facet object.

//3 We define a type for an output iterator to a stream buffer.

Time formatting facet objects write the formatted output via an iterator into an output container (see the sections on containers and iterators in the *User's Guide*). In principle, this can be an arbitrary container that has an output iterator, such as a string or a C++ array.

Here we want the time-formatting facet object to bypass the stream's formatting layer and write directly to the output stream's underlying stream buffer. Therefore, the output container shall be a stream buffer.

//4 We define a variable that will hold a reference to the locale object's `time_put` facet object. The time formatting facet class `time_put` has two template parameters:

The first template parameter is the character type used for output. Here we provide the stream's character type as the template argument.

The second template parameter is the output iterator type. Here we provide the stream buffer iterator type `outIter_t` that we had defined as before.

//5 Here we get the time-formatting facet object from the stream's locale via `use_facet()`.

//6 We define a variable to hold the output iterator returned by the facet object's formatting service.

//7 The facet object's formatting service `put()` is called. Let us see what arguments it takes. Here is the function's interface:

```
iter_type put      (iter_type             (a)
                     ,ios_base&             (b)
                     ,char_type             (c)
                     ,const tm*             (d)
                      char)                  (e)
```

The types `iter_type` and `char_type` stand for the types that were provided as template arguments when the facet class was instantiated. In this case, they are `ostreambuf_iterator<charT,traits>` and `charT`, where `charT` and `traits` are the respective streams template arguments.

Here is the actual call:

```
nextpos = fac.put(os,os,os.fill(),&date,'x');
```

Now let's see what the arguments mean:

a) The first parameter is supposed to be an output iterator. We provide an iterator to the stream's underlying stream buffer. The reference `os` to the output stream is converted to an output iterator, because output stream buffer iterators have a constructor taking an output stream, that is, `basic_ostream<charT,traits>&`.

b) The second parameter is of type `ios_base&`, which is one of the stream base classes. The class `ios_base` contains data for format control (see the section on iostreams for details). The facet object uses this formatting information. We provide the output stream's `ios_base` part here, using the automatic cast from a reference to an output stream, to a reference to its base class.

c) The third parameter is the fill character. It is used when the output has to be adjusted and blank characters have to be filled in. We provide the stream's fill character, which one can get by calling the stream's `fill()` function.

d) The fourth parameter is a pointer to a time structure `tm` from the C library.

e) The fifth parameter is a format character as in the C function `strftime()`; the `x` stands for the locale's appropriate date representation.

f) The value returned is an output iterator that points to the position immediately after the last inserted character.

//8 As we work with output stream buffer iterators, we can even check for errors happening during the time formatting. Output stream buffer iterators are the only iterators that have a member function `failed()` for error indication.[11]

//9 If there was an error, we set the stream's state accordingly. See the section on iostreams for details on the`setstate()` function and the state bits.

1.5.4 Creating a Facet Class for Replacement in a Locale

At times you may need to replace a facet object in a locale by another kind of facet object. In the following example, let us derive from one of the standard facet classes, `numpunct`, and create a locale object in which the standard `numpunct` facet object is replaced by an instance of our new, derived facet class.

Here is the problem we want to solve. When you print boolean values, you can choose between the numeric representation of the values `"true"` and `"false"`, or their alphanumeric representation.

```
int main(int argc, char** argv)
{
   bool any_arguments = (argc > 1);                          \\1
   cout.setf(ios_base::boolalpha);                           \\2
   cout << any_arguments << '\n';                            \\3
   // …
}
```

//1 A variable of type `bool` is defined. Its initial value is the boolean value of the logical expression `(argc > 1)`, so the variable `any_arguments` contains the information, whether the program was called with or without arguments.

---

[11] Note that the use of the put() function of a formatting facet is inherently unsafe, if you work with output iterators other than output stream buffer iterators. It is especially dangerous if you work with output iterators that refer to fixed-size containers, like a C++ array for example. There is no way to check whether the facet does not write beyond the containers end.

//2 The format flag ios_base:: boolalpha is set in the predefined output stream cout. The effect is that the string representation of boolean values is printed, instead of their numerical representation 0 or 1, which is the default representation.

//3 Here either the string "true" or the string "false" will be printed.

Of course, the string representation depends on the language. Hence, the alphanumeric representation of boolean values is provided by a locale. It is the numpunct facet of a locale that describes the cultural conventions for numerical formatting. It has a service that provides the string representation of the boolean values true and false.[12]

This is the interface of facet numpunct:

```
template <class charT>
    class numpunct : public locale::facet {
    public:
      typedef charT              char_type;
      typedef basic_string<charT> string_type;
      explicit numpunct(size_t refs = 0);
      string_type  decimal_point()   const;
      string_type  thousands_sep()   const;
      vector<char> grouping()        const;
      string_type  truename()        const;
      string_type  falsename()       const;
      static locale::id id;
};
```

Now let us replace this facet. To make it more exciting, let's use not only a different language, but also different words for true and false, such as Yes! and No!. For just using another language, we would not need a new facet; we would simply use the right native locale, and it would contain the right facet.

```
template <class charT, charT* True, charT* False>            //1
class CustomizedBooleanNames
: public numpunct_byname<charT> {                             //2
    typedef basic_string<charT> string;
  protected:
    string do_truename()  {return True;}                      //3
    string do_falsename() {return False;}
    ~CustomizedBooleanNames() {}
  public:
    explicit CustomizedBooleanNames(const char* LocName)      //4
         : numpunct_byname<charT>(LocName) {}
};
```

[12] You might be surprised to find the string representation of boolean values in the numpunct facet, because bool values are not numerical values. However, that's the way the facets are organized.

//1 The new facet is a class template that takes the character type as a template parameter, and the string representation for `true` and `false` as non-type template parameters.

//2 The new facet is derived from the `numpunct_byname<charT>` facet.

The byname facets read the respective locale information from the external representation of a C locale. The name provided to construct a byname facet is the name of a locale, as you would use it in a call to `setlocale()`.

//3 The virtual member functions `do_truename()` and `do_falsename()` are reimplemented. They are called by the public member functions `truename()` and `falsename()`. See the *Class Reference* for further details.

//4 A constructor is provided that takes a locale name. This locale's `numpunct` facet will be the basis for our new facet.

Now let's replace the `numpunct` facet object in a given locale object, as shown in the following figure:

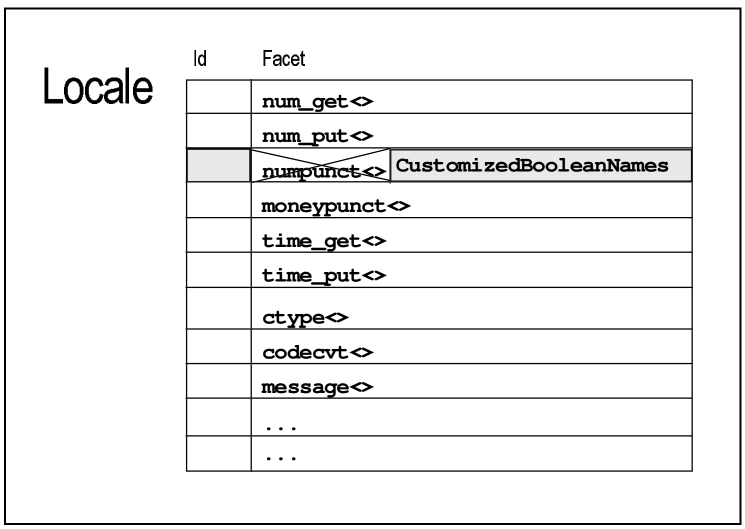


Figure 12. Replacing the numpunct facet object

The code looks like this:

```
char Yes[] = "Ja.";
char No[] = "Nein.";

void main(int argc, char** argv)
{
  locale loc(locale("de_DE"),                                      \\1
       new CustomizedBooleanNames<char,Yes,No>("de_DE"));        \\2
  cout.imbue(loc);                                                 \\3
  cout << "Argumente vorhanden? "  //Any arguments?
       << boolalpha << (argc > 1) << endl;                         \\4
```

```
}
```

//1 A locale object is constructed with an instance of the new facet class. The locale object will have all facet objects from a German locale object, except that the new facet object `CustomizedBooleanNames` will substitute for the `numpunct` facet object.

//2 The new facet object takes all information from a German `numpunct` facet object, and replaces the default native names `true` and `false` with the provided strings `"Ja."` ("Yes.") and `"Nein."` ("No.").

Note that the facet object is created on the heap. That's because the `locale` class by default manages installation, reference-counting, and destruction of all its facet objects.

//3 The standard output stream `cout` is imbued with the newly created locale object.

//4 The expression `(argc > 1)` yields a boolean value, which indicates whether the program was called with arguments. This boolean value's alphanumeric representation is printed to the standard output stream. The output might be:

```
Argument vorhanden? Ja.
```

1.5.5 The Facet Id

In the example discussed above, we derived a new facet class from one of the standard facet classes, then replaced an object of base class type by one of derived class type. The inheritance relationship of the facet classes is essential if you plan on replacing facet objects. Let us see why this is true.

A locale object maintains a set of facet objects. Each facet object has an identification that serves as an index to the set of facet objects. This identification, called `id`, is a static data member of the respective facet class. Whether or not a facet object will replace another facet, or be an actual addition to the locale object's set of facet objects, solely depends on the facet's identification.

The base class of all facets, class `locale::facet`, does not have a facet identification. The class `locale::facet` performs the function of an abstract base class; there will never be any facet object of the base class type. However, all concrete facet classes have to define a facet identification. In the example above, we inherited the facet identification from the base class we derived from, that is, the standard facet class `numpunct`. Every object of our facet class `CustomizedBooleanNames` has a facet identification that identifies it as a numpunct facet. As the facet identification serves as an index to the locale object's set of facets, our facet object replaced the current numpunct facet object in the locale object's set of facet objects.

If you do not want to replace a facet object, but want to add a new kind of facet object, we have to provide it with a facet identification different from all existing facet identifications. The following example will demonstrate how this can be achieved.

1.5.6 Creating a Facet Class for Addition to a Locale

At times you may need to add a facet object to a locale. This facet object must have a facet identification that distinguishes it from all existing kinds of facets.

Here is an example of a new facet class like that. It is a facet that checks whether a given character is a German umlaut [13], that is, one of the special characters äöüÄÖÜ.

```
class Umlaut : public locale::facet {                        \\1
  public:
    static locale::id id;                                    \\2
    bool is_umlaut(char c);                                  \\3
    Umlaut() {}
  protected:
    ~Umlaut() {}
};
```

//1 All facet classes have to be derived from class locale::facet.

//2 Here we define the static data member id. It is of type locale::id. The default constructor of the facet identification class locale::id assigns the next unused identification to each object it creates. Hence, it is not necessary, nor even possible, to explicitly assign a value to the static facet id object. In other words, this definition does the whole trick; our facet class will have a facet identification that distinguishes it from all other facet classes.

//3 A member function is_umlaut() is declared that returnstrue if the character is a German umlaut.

Now let's add the new facet object to a given locale object, as shown in the following figure:

---

[13] Generally, an umlaut is a composed character consisting of a vowel as the base character and a diaeresis, that is, two dots placed over a vowel, as the diacritic. The diaeresis is used in German and other European languages to indicate a change in the pronunciation of the vowel.

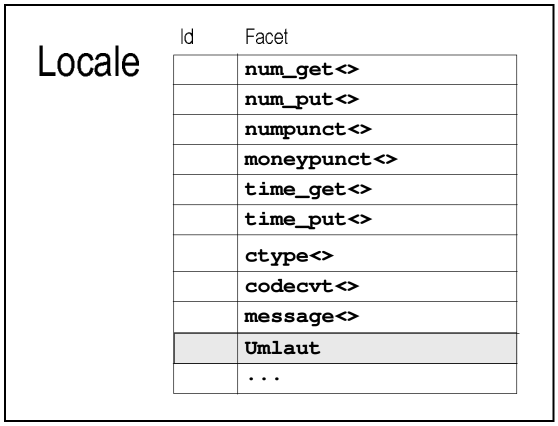


Figure 13. Adding a new facet to a locale

The code for this procedure is given below:

```
locale loc(locale(""),  // native locale
           new Umlaut); // the new facet                        //1
char c,d;
while (cin >> c){
 d = use_facet<ctype<char> >(loc).tolower(c);                    //2
 if (has_facet<Umlaut>(loc))                                     //3
 { if (use_facet<Umlaut>(loc).is_umlaut(d))                      //4
       cout << c << "belongs to the German alphabet!" << '\n';
 }
}
```

//1 A locale object is constructed with an instance of the new facet class. The locale object will have all facet objects from the native locale object, plus an instance of the new facet class `Umlaut`.

//2 Let's assume our new umlaut facet class is somewhat limited; it can handle only lower case characters. Thus we have to convert each character to a lower case character before we hand it over to the umlaut facet object. This is done by using a `ctype` facet object's service function `tolower()`.

//3 Before we use the umlaut facet object, we check whether such an object is present in the locale. In a toy example like this it is obvious, but in a real application it is advisable to check for the existence of a facet object, especially if it is a non-standard facet object we are looking for.

//4 The umlaut facet object is used, and its member function `is_umlaut()` is called. Note that the syntax for using this newly contrived facet object is exactly like the syntax for using the standard `ctype` facet.

1.6 User-Defined Facets: An Example

The previous sections explained how to use locales and the standard facet classes, and how you can build new facet classes. This section introduces you to the technique of building your own facet class and using it in conjunction with the input/output streams of the Standard C++ Library, the iostreams. This material is rather advanced, and requires some knowledge of standard iostreams.

In the following pages, we will work through a complete example on formatting telephone numbers. Formatting telephone numbers involves local conventions that vary from culture to culture. For example, the same US phone number can have all of the formats listed below:

754-3010	Local
(541) 754-3010	Domestic
+1-541-754-3010	International
1-541-754-3010	Dialed in the US
001-541-754-3010	Dialed from Germany
191 541 754 3010	Dialed from France

Now consider a German phone number. Although a German phone number consists of an area code and an extension like a US number, the format is different. Here is the same German phone number in a variety of formats:

636-48018	Local
(089) / 636-48018	Domestic
+49-89-636-48018	International
19-49-89-636-48018	Dialed from France

Note the difference in formatting domestic numbers. In the US, the convention is `1 (area code) extension`, while in Germany it is `(0 area code)/extension`.

1.6.1 A Phone Number Class

An application that has to handle phone numbers will probably have a class that represents a phone number. We will also want to read and write telephone numbers via iostreams, and therefore define suitable extractor and inserter functions. For the sake of simplicity, we will focus on the inserter function in our example.

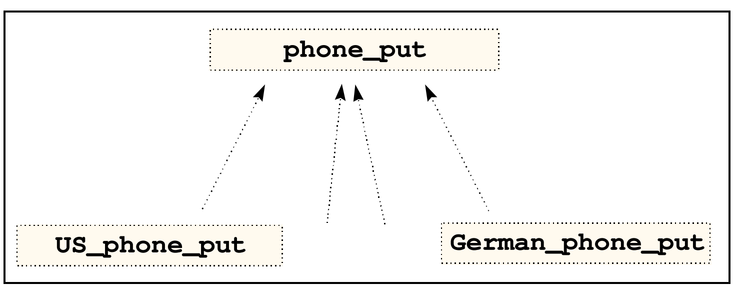
To begin, here is the complete class declaration for the telephone number class `phoneNo`:

```
class phoneNo
{
public:
 typedef basic_ostream<char> outStream_t;
 typedef string string_t;

 phoneNo(const string_t& cc,const string_t& ac,const string_t& ex)
    : countryCode(cc), areaCode(ac), extension(ex) {}

private:
   string_t countryCode;                                    //"de"
   string_t areaCode;                                       //"89"
   string_t extension;                              //"636-48018"

friend phoneNo::outStream_t& operator<<
   (phoneNo::outStream_t&, const phoneNo&);
};
```

1.6.2 A Phone Number Formatting Facet Class

Now that we have locales and facets in C++, we can encapsulate the locale-dependent parsing and formatting of telephone numbers into a new facet class. Let's focus on formatting in this example. We will call the new facet class `phone_put`, analogous to `time_put`, `money_put`, etc.

The `phone_put` facet class serves solely as a base class for facet classes that actually implement the locale-dependent formatting. The relationship of class`phone_put` to the other facet classes is



illustrated in the figure below:

Figure 14. The relationship of the `phone_put` facet to the implementing facets

Here is a first tentative declaration of the new facet class`phone_put`:

```
class phone_put: public locale::facet                       //1
{
public:
   static locale::id id;                                    //2
   phone_put(size_t refs = 0) : locale::facet(refs) { }     //3
```

```
    string_t put(const string_t& ext
                ,const string_t& area
                ,const string_t& cnt) const;                  //4
};
```

//1 Derive from the base class `locale::facet`, so that a locale object will be able to maintain instances of our new phone facet class.

//2 New facet classes need to define a static data member `id` of type `locale::id`.

//3 Define a constructor that takes the reference count that will be handed over to the base class.

//4 Define a function `put()` that does the actual formatting.

1.6.3 An Inserter for Phone Numbers

Now let's take a look at the implementation of the inserter for our phone number class:

```
ostream& operator<<(ostream& os, const phoneNo& pn)
{
 locale loc = os.getloc();                                    //1
 const phone_put& ppFacet = use_facet<phone_put> (loc);       //2
 os << ppFacet.put(pn.extension, pn.areaCode, pn.countryCode);//3
 return (os);
}
```

//1 The inserter function will use the output stream's locale object (obtained via `getloc()`),

//2 use the locale's phone number facet object,

//3 and call the facet object's formatting service `put()`.

1.6.4 The Phone Number Facet Class Revisited

Let us now try to implement the phone number facet class. What does this facet need to know?

- A facet needs to know its own locality, because a phone number is formatted differently for domestic and international use; for example, a German number looks like (089) / 636-48018 when used in Germany, but it looks like +1-49-89-636-48018 when used internationally.
- A facet needs information about the prefix for dialing international numbers; for example, 011 for dialing foreign numbers from the US, or 00 from Germany, or 19 from France.
- A facet needs access to a table of all country codes, so that one can enter a mnemonic for the country instead of looking up the

respective country code. For example, I would like to say: “This is a phone number somewhere in Japan” without having to know what the country code for Japan is.

1.6.4.1 Adding Data Members

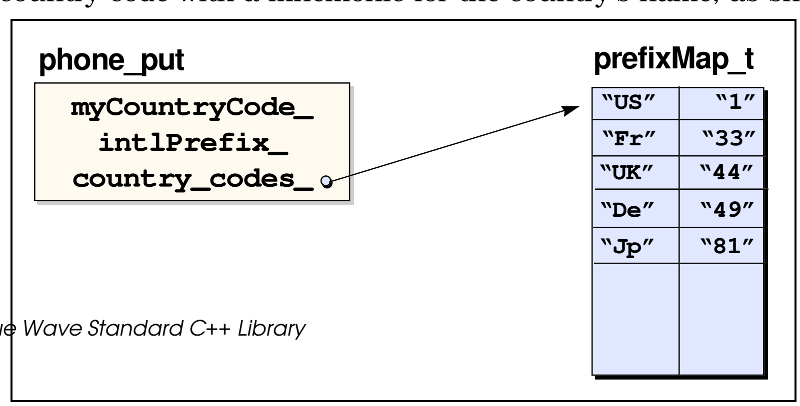
The following class declaration for the telephone number formatting facet class is enhanced with data members for the facet object's own locality, and its prefix for international calls (see //2 and //3 in the code below). Adding a table of country codes is omitted for the time being.

```
class phone_put: public locale::facet {
public:
  typedef string string_t;
  static locale::id id;
  phone_put(size_t refs = 0)  : locale::facet(refs)
                              , myCountryCode_("")
                              , intlPrefix_("")      {  }
 string_t put(const string_t& ext,
              const string_t& area,
              const string_t& cnt) const;
protected:
  phone_put( const string_t& myC                              //1
           , const string_t& intlP
           , size_t refs = 0)
           : locale::facet(refs)
           , myCountryCode_(myC)
           , intlPrefix_(intlP)   {  }
  const string_t myCountryCode_;                              //2
  const string_t intlPrefix_;                                 //3
};
```

Note how this class serves as a base class for the facet classes that really implement a locale-dependent phone number formatting. Hence, the public constructor does not need to be extended, and a protected constructor is added instead (see //1 above).

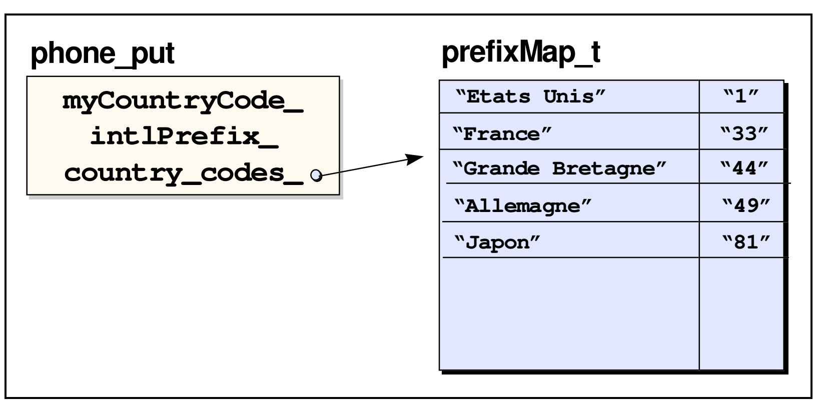
1.6.4.2 Adding Country Codes

Let us now deal with the problem of adding the international country codes that were omitted from the previous class declaration. These country codes can be held as a map of strings that associates the country code with a mnemonic for the country’s name, as shown



in the figure below:

Figure 15. Map associating country codes with mnemonics for countries' names



In the following code, we add the table of country codes:

```
class phone_put: public locale::facet
{
 public:
   class prefixMap_t : public map<string,string>             //1
   {
    public:
       prefixMap_t() { insert(tab_t(string("US"),string("1")));
       insert(tab_t(string("De"),string("49")));
       // ...
   }
};
   static const prefixMap_t* std_codes()                       //2
          { return &stdCodes_; }
 protected:
   static const prefixMap_t stdCodes_;                         //3
};
```

As the table of country codes is a constant table that is valid for all telephone number facet objects, it is added as a static data member `stdCodes_` (see `//3`). The initialization of this data member is encapsulated in a class, `prefixMap_t` (see `//1`). For convenience, a function `std_codes()` is added to give access to the table (see `//2`).

Despite its appealing simplicity, however, having just one static country code table might prove too inflexible. Consider that mnemonics might vary from one locale to another due to different languages. Maybe mnemonics are not called for, and you really need more extended names associated with the actual country code.

In order to provide more flexibility, we can build in the ability to work with an arbitrary table. A pointer to the respective country code table can be provided when a facet object is constructed. The static table, shown in the figure below, will serve as a default:

Figure 16. Map associating country codes with country names

Since we hold the table as a pointer, we need to pay attention to memory management for the table pointed to. We will use a flag for determining whether the provided table needs to be deleted when the facet is destroyed. The following code demonstrates use of the table and its associated flag:

```
class phone_put: public locale::facet {
public:
  typedef string string_t;
  class prefixMap_t;
  static locale::id id;

  phone_put( const prefixMap_t* tab=0                        //1
           , bool del = false
           , size_t refs = 0)
           : locale::facet(refs)
           , countryCodes_(tab), delete_it_(del)
           , myCountryCode_(""), intlPrefix_("")
  { if (tab)   {  countryCodes_ = tab;
                  delete_it_ = del;   }
    else       {  countryCodes_ = &stdCodes_;                //2
                  delete_it_ = false; }
  }
  string_t put(const string_t& ext,
               const string_t& area,
               const string_t& cnt) const;

  const prefixMap_t* country_codes() const                   //3
  { return countryCodes_; }

  static const prefixMap_t* std_codes()    { return &stdCodes_; }
protected:
  phone_put(const string_t& myC, const string_t& intlP
          , const prefixMap_t* tab=0, bool del = false
          , size_t refs = 0)
          : locale::facet(refs)
          , countryCodes_(tab), delete_it_(del)
          , myCountryCode_(myC), intlPrefix_(intlP)
  { ... }
  virtual ~phone_put()
  { if(delete_it_)
      countryCodes_->prefixMap_t::~prefixMap_t();            //4
  }

  const prefixMap_t* countryCodes_;                          //5
  bool delete_it_;
  static const prefixMap_t stdCodes_;
  const string_t myCountryCode_;
  const string_t intlPrefix_;
};
```

//1 The constructor is enhanced to take a pointer to the country code table, together with the flag for memory management of the provided table.

//2 If no table is provided, the static table is installed as a default.

//3 For convenience, a function that returns a pointer to the current table is added.

//4 The table is deleted if the memory management flags says so.

//5 Protected data members are added to hold the pointer to the current country code table, as well as the associated memory management flag.

1.6.5 An Example of a Concrete Facet Class

As mentioned previously, the phone number facet class is intended to serve as a base class. Let's now present an example of a concrete facet class, the US phone number formatting facet. It works by default with the static country code table and "US" as its own locality. It also knows the prefix for dialing foreign numbers from the US. Here is the class declaration for the facet:

```
class US_phone_put : public phone_put {
public:
   US_phone_put( const prefixMap_t* tab=0
               , const string_t& myCod = "US"
               , bool del = false
               , size_t refs = 0)
               : phone_put(myCod,"011",tab,del,refs)
   { }
};
```

Other concrete facet classes are built similarly.

1.6.6 Using Phone Number Facets

Now that we have laid the groundwork, we will soon be ready to format phone numbers. Here is an example of how instances of the new facet class can be used:

```
ostream ofstr("/tmp/out");
ostr.imbue(locale(locale::classic(),new US_phone_put));       //1
ostr << phoneNo("Fr","1","60 17 07 16") << endl;
ostr << phoneNo("US","541","711-PARK") << endl;

ostr.imbue(locale(locale("Fr")                                //2
          ,new Fr_phone_put (&myTab,"France")));
ostr << phoneNo("Allemagne","89","636-40938") << endl;        //3
```

//1 Imbue an output stream with a locale object that has a phone number facet object. In the example above, it is the US English ASCII locale with a US phone number facet, and

//2 a French locale using a French phone number facet with a particular country code table.

//3 Output phone numbers using the inserter function.

The output will be: 011-33-1-60170716
(541) 711-PARK
19 49 89 636 40938

1.6.7 Formatting Phone Numbers

Even now, however, the implementation of our facet class is incomplete. We still need to mention how the actual formatting of a phone number will be implemented. In the example below, it is done by calling two virtual functions, `put_country_code()` and `put_domestic_area_code()`:

```
class phone_put: public locale::facet {
public:
  // …
  string put(const string& ext,
             const string& area,
             const string& cnt) const;
protected:
  // …
  virtual string_t put_country_code
          (const string_t& country) const = 0;
  virtual string_t put_domestic_area_code
          (const string_t& area) const = 0;
};
```

Note that the functions`put_country_code()` and `put_domestic_area_code()` are purely virtual in the base class, and thus must be provided by the derived facet classes. For the sake of brevity, we spare you here the details of the functions of the derived classes. For more information, please consult the directory of sample code delivered on disk with this product.

1.6.8 Improving the Inserter Function

Let's turn here to improving our inserter function. Consider that the country code table might be huge, and access to a country code might turn out to be a time-consuming operation. We can optimize the inserter function's performance by caching the country code table, so that we can access it directly and thus reduce performance overhead.

1.6.8.1 Primitive Caching

The code below does some primitive caching. It takes the phone facet object from the stream's locale object and copies the country code table into a static variable.

```
ostream& operator<<(ostream& os, const phoneNo& pn)
{
   locale loc = os.getloc();
   const phone_put& ppFacet = use_facet<phone_put> (loc);

   // primitive caching
   static prefixMap_t codes = *(ppFacet.country_codes());

   // some sophisticated output using the cached codes
   ...
   return (os);
}
```

Now consider that the locale object imbued on a stream might change, but the cached static country code table does not. The cache is filled once, and all changes to the stream's locale object have no effect on this inserter function's cache. That's probably not what we want. What we do need is some kind of notification each time a new locale object is imbued, so that we can update the cache.

1.6.8.2 Registration of a Callback Function

In the following example, notification is provided by a callback function. The iostreams allow registration of callback functions. Class `ios_base` declares:

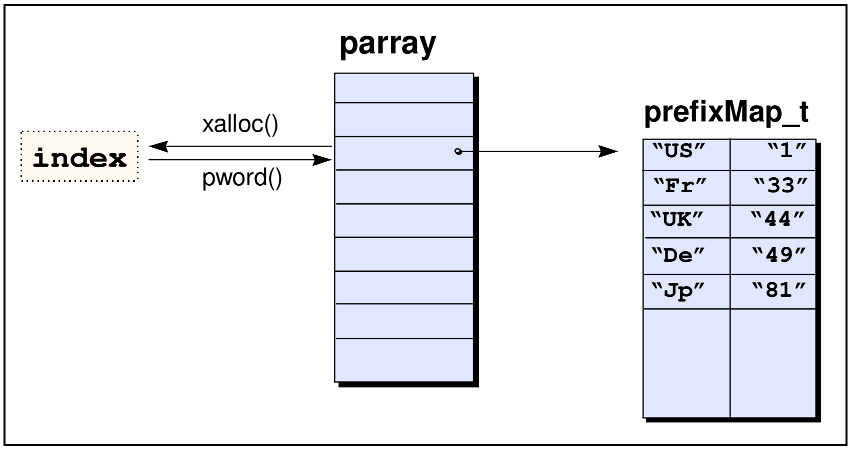
```
enum event { erase_event, imbue_event, copyfmt_event };       //1
typedef void (*event_callback) (event, ios_base&, int index);
void register_callback (event_callback fn, int index);        //2
```

`//1` Registered callback functions are called for three events:

- Destruction of a stream,
- Imbuing a new locale, and
- Copying the stream state.

//1 The `register_callback()` function registers a callback function and an index to the stream's `parray`. During calls to `imbue()`, `copyfmt()` or `~ios_base()`, the function `fn` is called with argument `index`. Functions registered are called when an event occurs, in opposite order of registration.

The `parray` is a static array in base class `ios_base`. One can obtain an index to this array via `xalloc()`, and access the array via `pword(index)` or `iword(index)`, as shown in the figure



below:

Figure 17. The static array parray

In order to install a callback function that updates our cache, we implement a class that retrieves an index to `parray` and creates the cache, then registers the callback function in its constructor. The procedure is shown in the code below:

```
class registerCallback_t {
public:
registerCallback_t(ostream& os
                  ,ios_base::event_callback fct
                  ,prefixMap_t* codes)
  {
   int index = os.xalloc();                                   //1
   os.pword(index) = codes;                                   //2
   os.register_callback(fct,index);                           //3
  }
};
```

//1 An index to the array is obtained via `xalloc()`.

//2 The pointer to the code table is stored in the array via `pword()`.

//3 The callback function and the index are registered.

The actual callback function will later have access to the cache via the index to parray.

At this point, we still need a callback function that updates the cache each time the stream's locale is replaced. Such a callback function could look like this:

```
void cacheCountryCodes(ios_base::event event
                      ,ios_base& str,int cache)
{  if (event == ios_base::imbue_event)                   //1
   {
      locale loc = str.getloc();
      const phone_put<char>& ppFacet =
                   use_facet<phone_put<char> > (loc);      //2

      *((phone_put::prefixMap_t*) str.pword(cache)) =
                   *(ppFacet.country_codes());             //3
   }
}
```

//1 It checks whether the event was a change of the imbued locale,

//2 retrieves the phone number facet from the stream's locale, and

//3 stores the country code table in the cache. The cache is accessible via the stream's parray.

1.6.8.3 Improving the Inserter

We now have everything we need to improve our inserter. It registers a callback function that will update the cache whenever necessary. Registration is done only once, by declaring a static variable of class registerCallback_t.

```
ostream& operator<<(ostream& os, const phoneNo& pn)
{
 static phone_put::prefixMap_t codes =
    *(use_facet<phone_put>(os.getloc()).country_codes());     //1

 static registerCallback_t cache(os,cacheCountryCodes,&codes);//2

  // some sophisticated output using the cached codes
  ...
}
```

//1 The current country code table is cached.

//2 The callback function cacheCountryCodes is registered.