Rogue Wave Standard C++ Library Iostreams and Locale User's Guide

(Part A)

Rogue Wave Software
Corvallis, Oregon USA

Rogue Wave Standard C++ Library Iostreams and Locale User's Guide and Reference

for

Rogue Wave's implementation of the Standard C++ Library.

Based on ANSI's Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++.

User's Guide and Tutorial Author: Timothy A. Budd

Internationalization and Locale User's Guide Author: Angelika Langer, with Elaine Cull

Class Reference Authors: Wendi Minne, Tom Pearson, and Randy Smithey

Product Team:

Development:	Anna Dahan, Angelika Langer, Philippe Le Mouel, Randy Smithey
Quality Engineering:	Kevin Djang, Randall Robinson
Manuals:	Elaine Cull, Wendi Minne, Julie Prince, Randy Smithey
Support:	North Krimsley

Significant contributions by: Joe Delaney

Printed in the United States of America.

Part # RW81-01-100096

Printing Date: October, 1996

Rogue Wave Software, Inc., 850 SW 35th St., Corvallis, Oregon, 97333 USA

Product Information:	**(541) 754-3010**
	(800) 487-3217
Technical Support:	**(541) 754-2311**
FAX:	**(541) 757-6650**
World Wide Web:	**http://www.roguewave.com**

Table of Contents

NOTE: See Part B for the Locale & Iostreams Reference Section (listed alphabetically)

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 availabity 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, Deutsch, Franҫais, Italiano, 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 ·A–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

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

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 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

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

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, 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.

Figure 1 gives an example of an encoded Japanese sentence composed of these four writing systems:

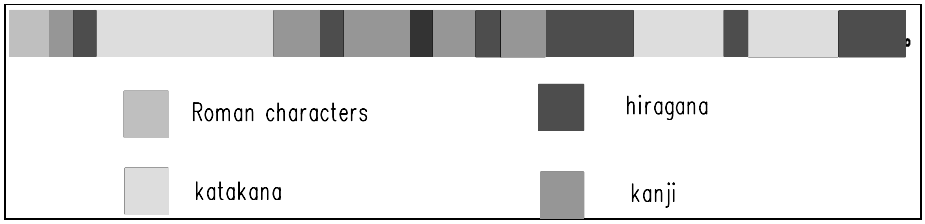


Figure 1. A Japanese sentence mixing four writing systems

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 Figure 2.

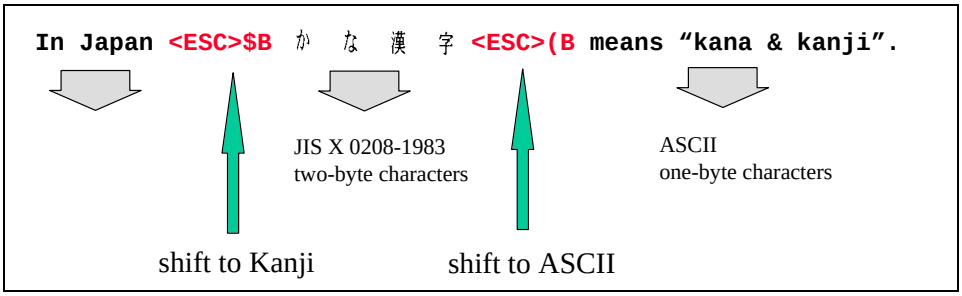


Figure 2. An example of a Japanese text encoded in JIS

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 e-mail 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 with adjacent names separated by a slash, like `/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 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.

---

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

There are many wide character standards, including those shown below:

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

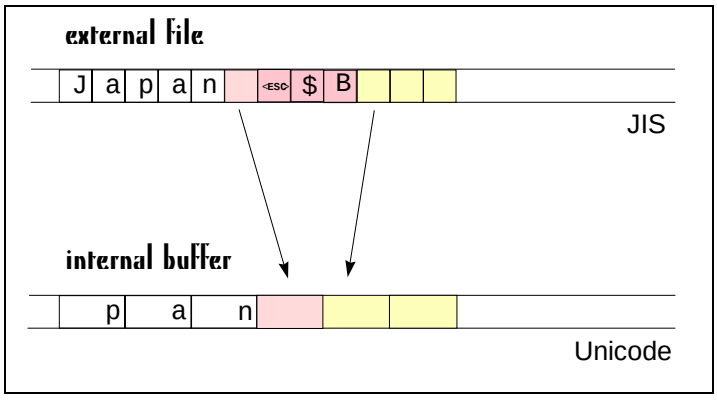
The programming language C++ supports wide characters; their native type in C++ is called `whar_t`. The syntax for wide character constants and wide character strings is similar to that for ordinary, tiny character constants and strings:

`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 progam, 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. Figure 3 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.*

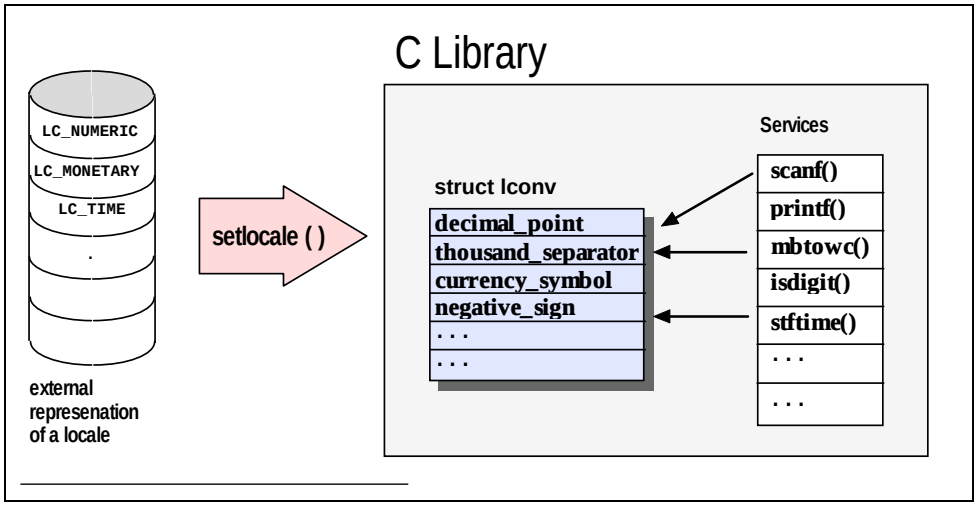
The X/Open consortium has standardized a variety of services for *Native Language Support (NLS)* in the programming language C. This standard is commonly known as XPG4. The X/Open's *Native Language Support* includes

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 Figure 4 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 in Table 2:

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. In fact, you can think of a C++ locale as a container of facets, as illustrated in Figure 5 below:

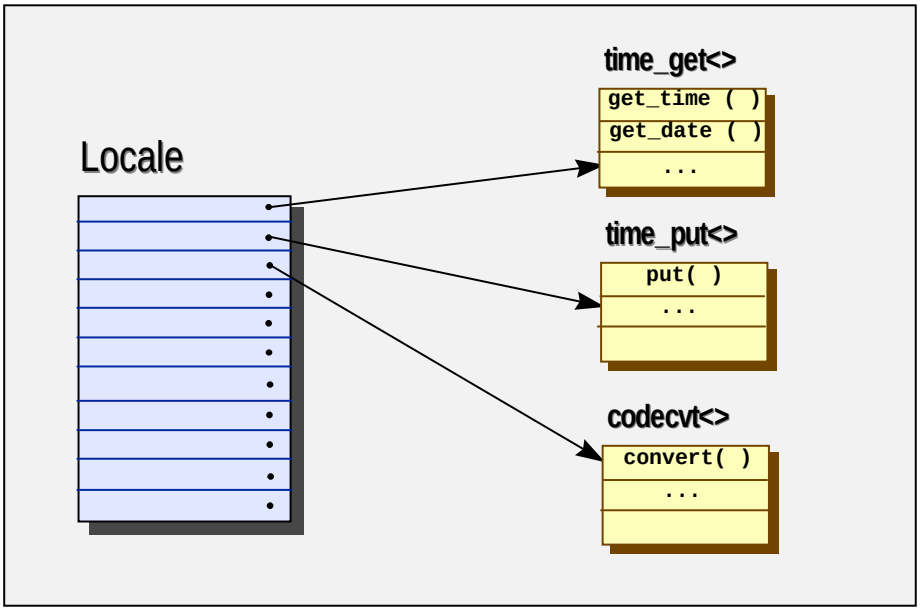


Figure 5. A C++ locale is a container of facets

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 could bundle additional internationalization services into a new facet class, or purchase a facet library.

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 classees `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, 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 must be printed in the customer's native language, so the application must 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. Figure 6 summarizes this activity:

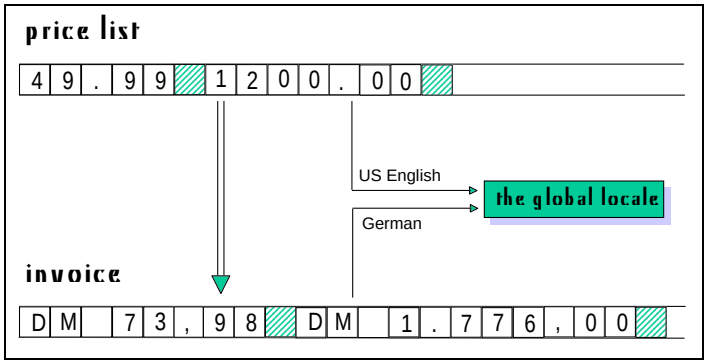


Figure 6. Multiple locales in C

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 *Figure 7*:

---

[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.

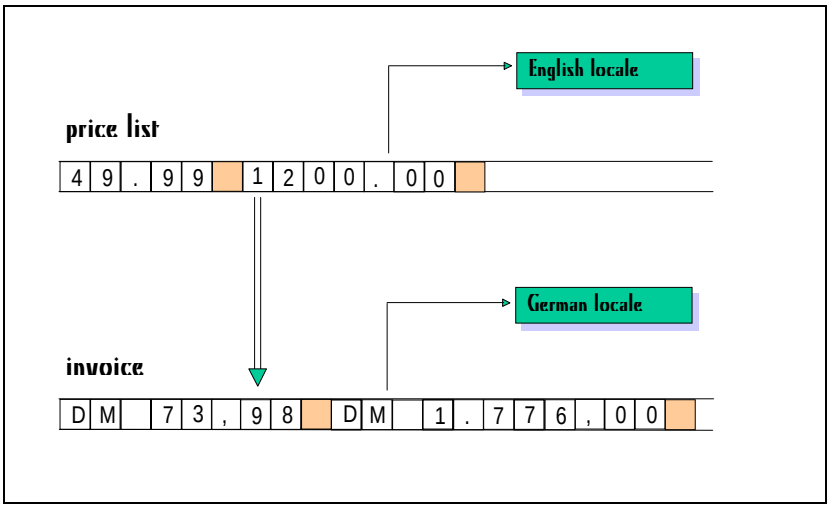


Figure 7. Multiple locales in C++

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:

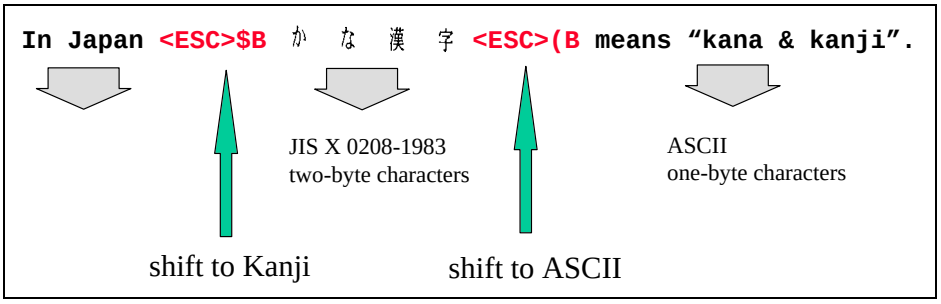


Figure 8. The Japanese text encoded in JIS from Figure 2

Suppose you are parsing input from a multibyte file which contains text that is encoded in JIS, as shown in Figure 9. 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.

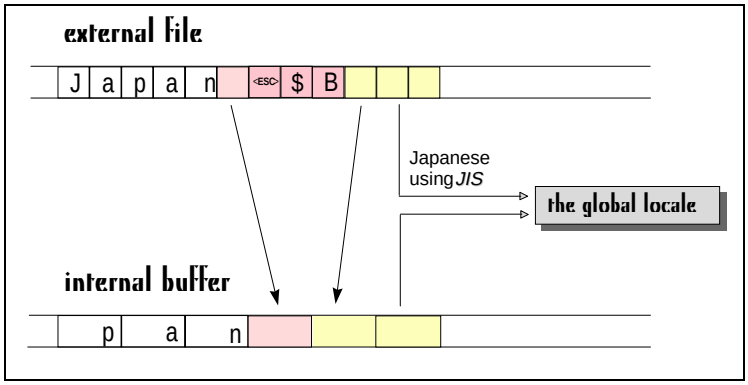


Figure 9. Parsing input from a multibyte file using the global C locale

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 which encapsulate 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.

Figure 10 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.

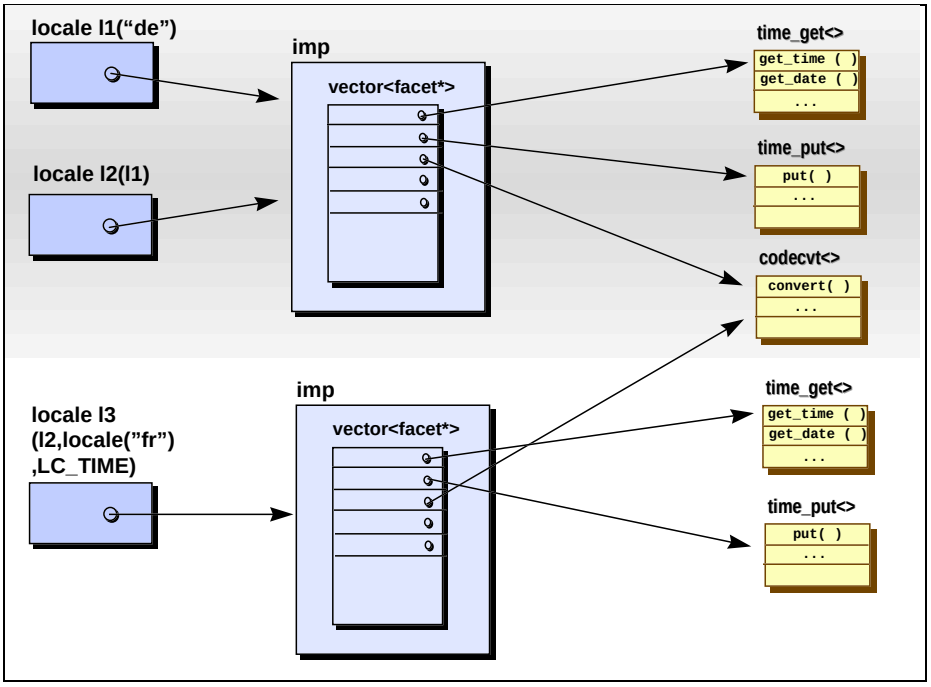


Figure 10. The locale architecture

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

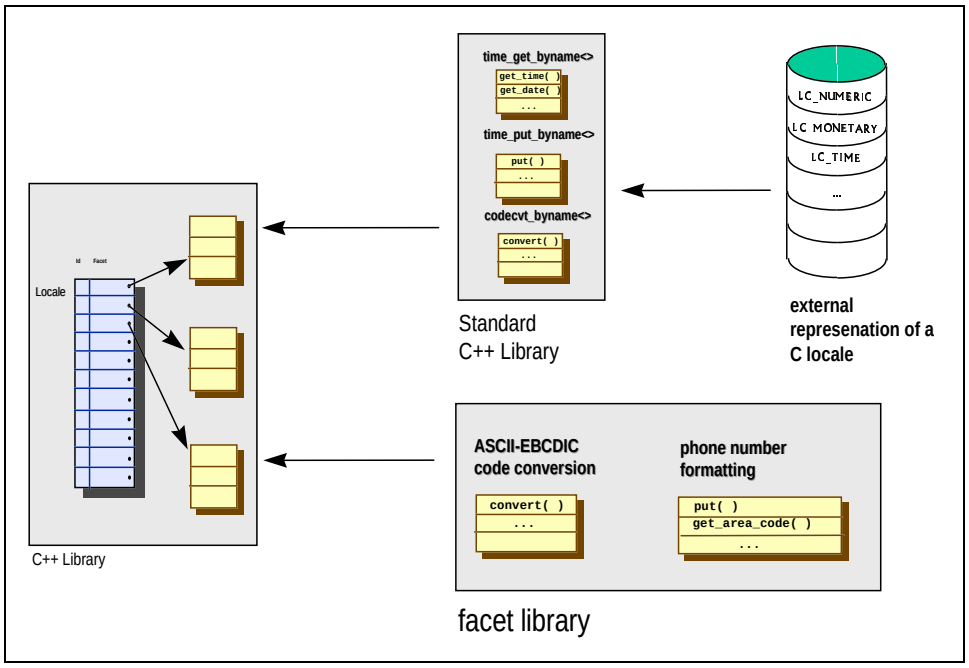
1.5 The Facets

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 Figure 11:

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 are not

---

[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.

used independently of each other, but grouped together in a locale object. For this reason, facet objects are usually constructed along with the locale object that maintains them.

On rare occasions, however, you may need to construct a single facet for stand-alone use. You will then find that you cannot construct a facet directly, because the `facet` class has a protected destructor. [10] The example below demonstrates how to write the code to construct and use a single facet object. This code demonstrates a locale-sensitive string comparison, which you would perform in C using the `strcoll()` function.

```
template <class charT>
class Collate : public collate_byname<charT>
{
public:
   Collate(const char* name, size_t refs=0)
   : collate_byname<charT>(name,refs) {}
   ~Collate() {}
};

string name1("Peter Gartner");
string name2 ("Peter Gärtner");
Collate<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 from a German C locale's external representation.

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

The string class in the Standard C++ Library does not provide any service for locale-sensitive string comparisons. Hence, you will generally use a collate facet's compare service, as demonstrated above, or the locale's function call operator instead:

```
string name1("Peter Gartner");
string name2 ("Peter Gärtner");
locale loc("De_DE");
if ( loc(name1, name2) )
{ … }
```

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

---

[10] The destructor is inaccessible to the public because it is assumed that a locale owns its facets and manages their storage.

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 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, however, 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
 nextpos = 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
   // …
}
```

---

[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.

`//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.

`//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
```

---

[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.

```
        : numpunct_byname<charT>(LocName) {}
};
```

`//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 Figure 12:

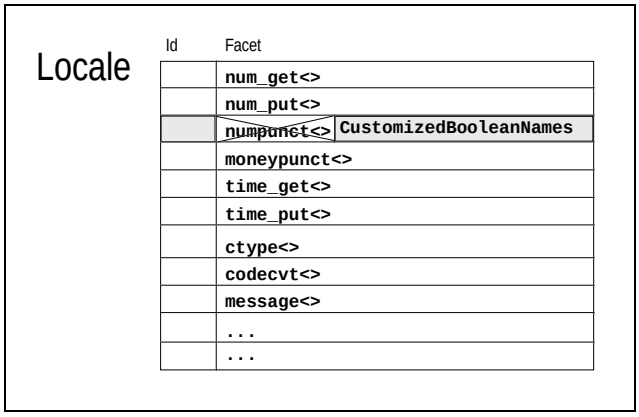


Figure 12. Replacing the numpunct facet object

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 returns `true` if the character is a German umlaut.

[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.

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

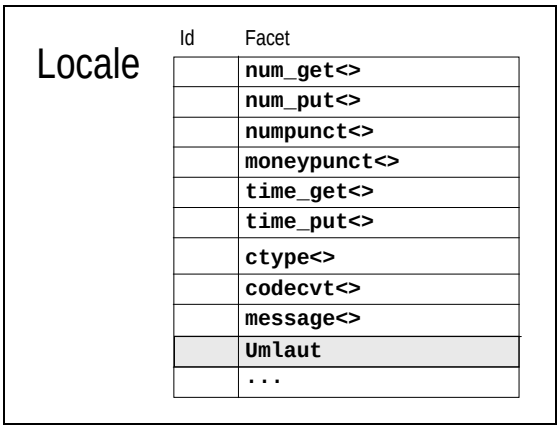


Figure 13:

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 Figure 14:

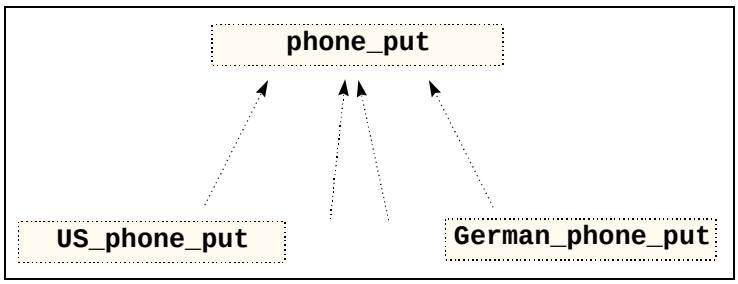


Figure 14. The relationship of the `phone_put` facet to the implementing facets

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 Figure 15:

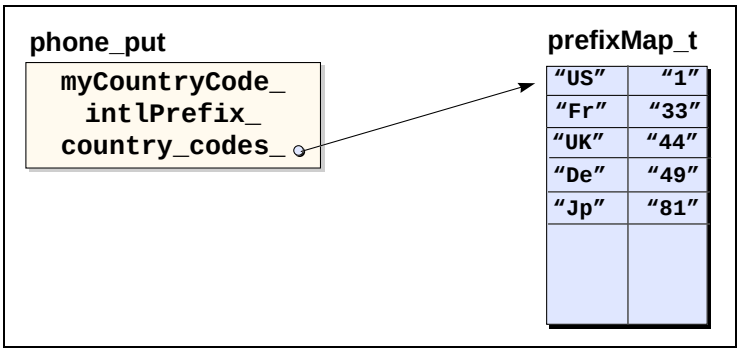


Figure 15. Map associating country codes with mnemonics for countries' names

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 Figure 16 below, will serve as a default:

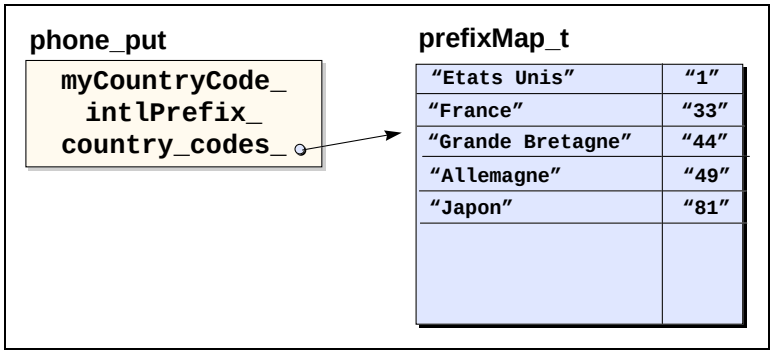


Figure 16. Map associating country codes with country names

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.

//2 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 Figure 17 below:

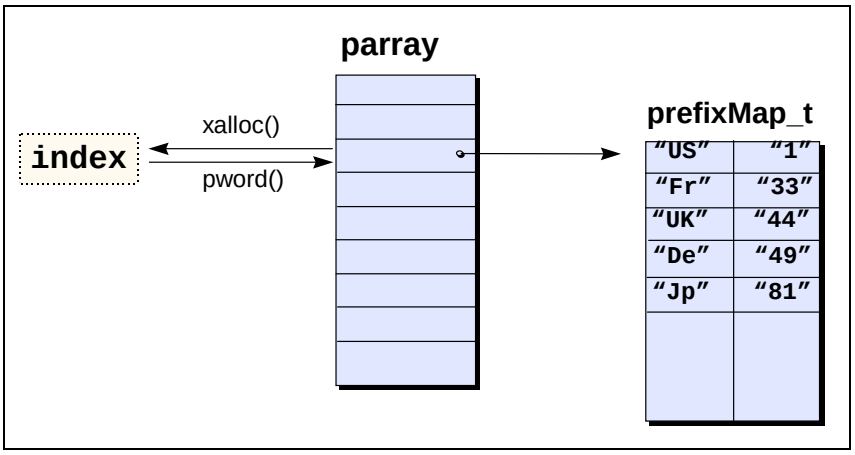


Figure 17. The static array parray

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.

Section 2.

Stream Input/Output

2.1 How to Read This Section

This section is an introduction to C++ stream input and output. Section 2.2 explains the iostreams facility, how it works in principle, and how it should be used. This section should be read by anyone who needs basic information on iostreams. For readers who require deeper understanding, the section also gives an overview of the iostream architecture, its components, and class hierarchy.

It is not necessary to read Section 2.2 in order to start using predefined iostreams as explained in Section 2.3. However, we do recommend that you read the *Users Guide*'s section on internationalization first, because iostreams is internationalized using standard C++ locales. The explanations in this section assume that you are familiar with locales and facets, and you will find countless references to the section on internationalization.

Sections 2.3 to 2.6 explain the basic operation of iostreams. These sections can be read sequentially, or used as a reference for certain topics. Read sequentially, they provide you with basic knowledge for working with iostreams. Used as a reference, they provide answers to questions like: *How do I check for iostreams errors?* and, *How do I work with file streams?*

Sections 2.7 to 2.11 explain simple techniques for extending the iostreams framework, such as defining input and output operators for user-defined types, and adding manipulators. These sections also cover more advanced features of iostreams, such as synchronization of streams.

Sections 2.12 and 2.13 explain advanced techniques for extending iostreams, such as creating new types of streams by derivation and defining a code conversion facet.

Section 2.14 describes the main differences between the Standard C++ Library iostreams and traditional iostreams.

Section 2.15 describes the main differences between the Standard C++ Library iostreams and the Rogue Wave implementation of iostreams in its own *Standard C++ Library*. It points out features that are specific to the Rogue Wave implementation.

The Appendix describes standardization issues that are still open at the time of this writing and influence the content of this document.

2.1.1 Code Examples

Please note that the examples in this *User's Guide* might not compile in your particular environment due to incompatibilities with the particular release of your compiler or your library. This is because few compilers available at the time of this printing are capable of understanding the whole range of language features defined by the ISO / ANSI C++ standard. It is likely that at

least a few of these features will not be supported by your own compiler. The consequence is that some techniques demonstrated and explained in this *User's Guide* will not work with your compiler either.

We include examples that might not compile, rather than omitting certain techniques entirely, to demonstrate the full range of techniques the Standard C++ language will support. This *User's Guide* was written with an eye to the C++ of the future. Compilers will catch up, and techniques that don't work with your current compiler *will* work once your compiler can understand Standard C++. Hopefully, including these techniques will extend the usefulness of this *User's Guide* to you.

Also, the code examples are simplified in that the necessary `#include <...>` statements and the using directive for the standard namespace `::std` are omitted. The intent is to make the examples as readable and focused as possible rather than ceaselessly repeating the same code fragments.

2.1.2 Terminology

The Standard C++ Library consists mostly of class and function templates. Abbreviations for these templates are used throughout this *User's Guide*. For example, `fstream` stands for `template <class charT, class traits> class basic_fstream`. A slightly more succinct notation for a class template is also frequently used: `basic_fstream <charT, traits>`.

In addition to abbreviations, you will find certain contrived technical terms. For example, *file stream* stands for the abstract notion of the file stream class template; `badbit` stands for the state flag `ios_base::badbit`.

2.2 The Architecture of Iostreams

This section will introduce you to iostreams: what they are, how they work, what kinds of problems they help solve, and how they are structured. Section 2.2.4 provides an overview of the class templates in iostreams. If you want to skip over the software architecture of iostreams, please go on to Section 2.3 on formatted input/output.

2.2.1 What Are the Standard Iostreams?

The Standard C++ Library includes classes for text stream input/output. Before the current ANSI/ISO standard, most C++ compilers were delivered with a class library commonly known as the *iostreams* library. In this section, we refer to this library as the *traditional iostreams*, in contrast to the *standard iostreams* that are now part of the ANSI/ISO Standard C++ Library. The standard iostreams are to some extent compatible with the traditional iostreams, in that the overall architecture and the most commonly used interfaces are retained. Section 2.14 describes the incompatibilities in greater detail.

We can compare the standard iostreams not only with the traditional C++ iostreams library, but also with the I/O support in the Standard C Library. Many former C programmers still prefer the input/output functions offered by the C library, often referred to as *C* stdio. Their familiarity with the C library is justification enough for using the C stdio instead of C++ iostreams, but there are other reasons as well. For example, calls to the C functions `printf()` and `scanf()` are admittedly more concise with C stdio. However, C stdio has drawbacks, too, such as type insecurity and inability to extend consistently for user-defined classes. We'll discuss these in more detail in the following sections.

2.2.1.1 Type Safety

Let us compare a call to stdio functions with the use of standard iostreams. The stdio call reads as follows:

```
int i = 25;
char name[50] = "Janakiraman";
fprintf(stdout, "%d %s", i, name);
```

It correctly prints: `25 Janakiraman`.

But what if we inadvertently switch the arguments to `fprintf`? The error will be detected no sooner than run time. Anything can happen, from peculiar output to a system crash. This is not the case with the standard iostreams:

```
cout << i << ' ' << name << '\n';
```

Since there are overloaded versions of the shift operator `operator<<()`, the right operator will always be called. The function `cout << i` calls `operator<<(int)`, and `cout << name` calls `operator<<(const char*)`. Hence, the standard iostreams are typesafe.

2.2.1.2 Extensibility to New Types

Another advantage of the standard iostreams is that user-defined types can be made to fit in seamlessly. Consider a type `Pair` that we want to print:

```
struct Pair { int x; string y; }
```

All we need to do is overload `operator<<()` for this new type `Pair`, and we can output pairs this way:

```
Pair p(5, "May");
cout << p;
```

The corresponding `operator<<()` can be implemented as:

```
basic_ostream<char>&
operator<<(basic_ostream<char>& o, const Pair& p)
{ return o << p.x << ' ' << p.y; }
```

2.2.2 How Do the Standard Iostreams Work?

The main purpose of the standard iostreams is to serve as a tool for input and output of text. Generally, input and output are the transfer of data between a program and any kind of external device, as illustrated in Figure 18 below:

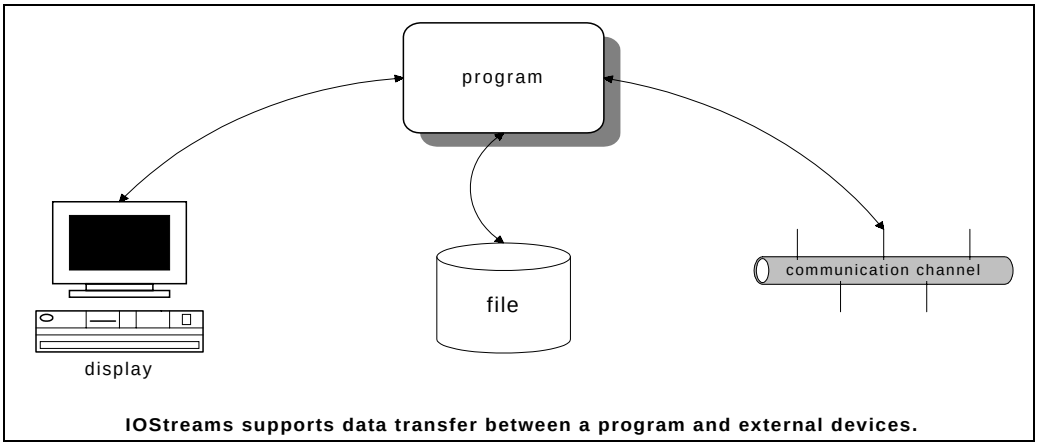


Figure 18. Data transfer supported by iostreams

The internal representation of such data is meant to be convenient for data processing in a program. On the other hand, the external representation can vary quite a bit: it might be a display in human-readable form, or a portable data exchange format. The intent of a representation, such as conserving space for storage, can also influence the representation.

T*ext I/O* involves the external representation of a sequence of characters; every other case involves *binary I/O*. Traditionally, iostreams are used for text processing. Such text processing through iostreams involves two processes: *formatting* and *code conversion*.

Formatting is the transformation from a byte sequence representing internal data into a human-readable character sequence; for example, from a floating point number, or an integer value held in a variable, into a sequence of digits. Figure 19 below illustrates the formatting process:

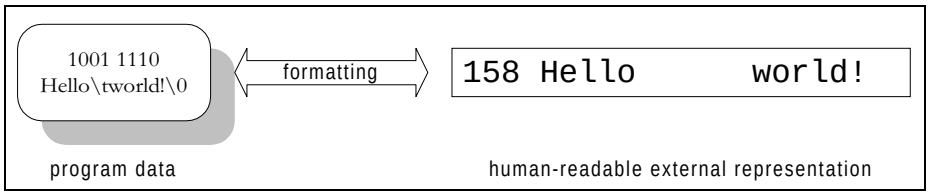


Figure 19. Formatting program data

Code conversion is the process of translating one character representation into another; for example, from wide characters held internally to a sequence of multibyte characters for external use. Wide characters are all the same size, and thus are convenient for internal data processing. Multibyte characters have different sizes and are stored more compactly. They are typically used for data transfer, or for storage on external devices such as files. Figure 20 below illustrates the conversion process:

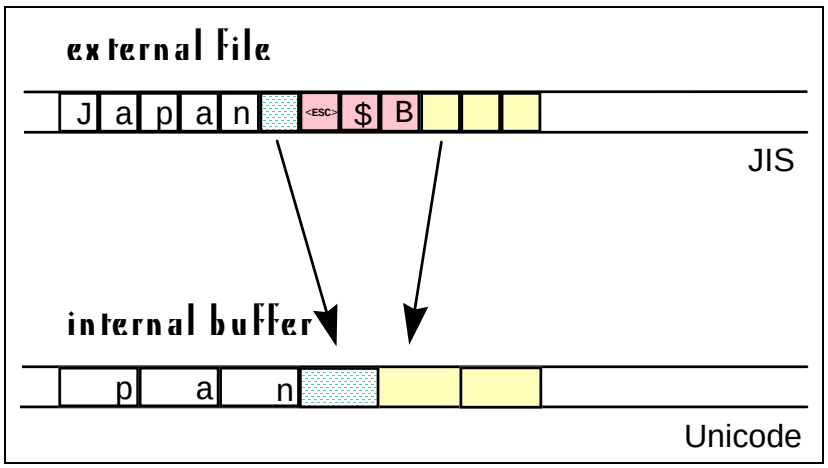


Figure 20. Code conversion between multibytes and wide characters

2.2.2.1 The Iostream Layers

The iostreams facility has two layers: one that handles formatting, and another that handles code conversion and transport of characters to and from the external device. The layers communicate through a buffer, as illustrated in Figure 21 below:

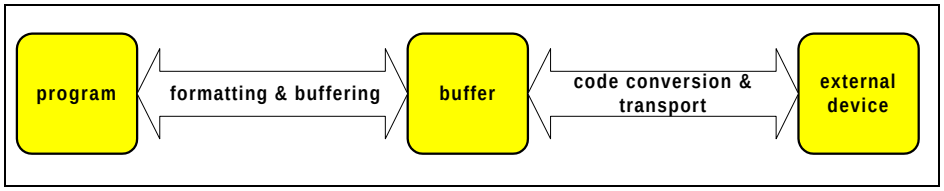


Figure 21. The iostreams layers

Let's take a look at the function of each layer in more detail:

- **The Formatting Layer**. Here the transformation between a program's internal data representation and a readable representation as a character sequence takes place. This formatting and parsing may involve, among other things:

- Precision and notation of floating point numbers;
- Hexadecimal, octal, or decimal representation of integers;
- Skipping of white space in the input;
- Field width for output;
- Adapting of number formatting to local conventions.

- **The Transport Layer**. This layer is responsible for producing and consuming characters. It encapsulates knowledge about the properties of a specific external device. Among other things, this involves:
  - Block-wise output to files through system calls;
  - Code conversion to multibyte encodings.

  To reduce the number of accesses to the external device, a buffer is used. For output, the formatting layer sends sequences of characters to the transport layer, which stores them in a *stream buffer*. The actual transport to the external device happens only when the buffer is full. For input, the transport layer reads from the external device and fills the buffer. The formatting layer receives characters from the buffer. When the buffer is empty, the transport layer is responsible for refilling it.

- **Locales**. Both the formatting and the transport layers use the stream's locale. (See the section on internationalization and locales.) The formatting layer delegates the handling of numeric entities to the locale's numeric facets. The transport layer uses the locale's code conversion facet for character-wise transformation between the buffer content and characters transported to and from the external device. Figure 22 below shows how locales are used with iostreams:

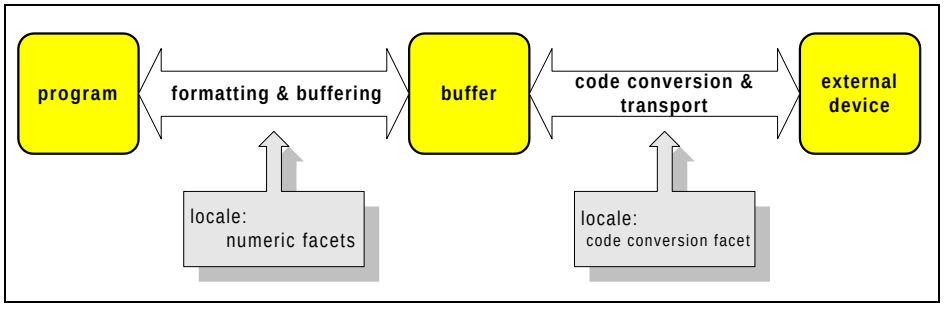


Figure 22. Use of locales in iostreams

2.2.2.2 File and In-Memory I/O

Iostreams support two kinds of I/O: *file I/O* and *in-memory I/O*.

File I/O involves the transfer of data to and from an external device. The device need not necessarily be a file in the usual sense of the word. It could

just as well be a communication channel, or another construct that conforms to the file abstraction.

In contrast, in-memory I/O involves no external device. Thus code conversion and transport are not necessary; only formatting is performed. The result of such formatting is maintained in memory, and can be retrieved in the form of a character string.

2.2.3 How Do the Standard Iostreams Help Solve Problems?

There are many situations in which iostreams are useful:

- **File I/O**. Iostreams can still be used for input and output to files, although file I/O has lost some of it former importance. In the past, alpha-numeric user-interfaces were often built using file input/output to the standard input and output channels. Today almost all applications have graphical user interfaces.

  Nevertheless, iostreams are still useful for input and output to files other than the standard input and output channels, and to all other kinds of external media that fit into the file abstraction. For example, the Rogue Wave class library for network communications programming, *Net.h++*, uses iostreams for input and output to various kinds of communication streams like sockets and pipes.

- **In-Memory I/O.** Iostreams can perform in-memory formatting and parsing. Even with a graphical user interface, you have to format the text you want to display. The standard iostreams offer internationalized in-memory I/O, which is a great help for text processing tasks like formatting. The formatting of numeric values, for example, depends on cultural conventions. The formatting layer uses a locale's numeric facets to adapt its formatting and parsing to cultural conventions.

- **Internationalized Text Processing.** This function is actively supported by iostreams.

  Iostreams use locales. As locales are extensible, any kind of facet can be carried by a locale, and thus used by a stream. By default, iostreams use only the numeric and the code conversion facets of a locale. However, date , time, and monetary facets are available in the Standard C++ Library. Other cultural dependencies can be encapsulated in unique facets and made accessible to a stream. You can easily internationalize your use of iostreams to meet your needs.

- **Binary I/O.** The traditional iostreams suffer from a number of limitations. The biggest is the lack of conversion abilities: if you insert a `double` into a stream, for example, you do not know what format will be used to represent this `double` on the external device. There is no portable way to insert it as binary.

Standard iostreams are by far more flexible. The code conversion performed on transfer of internal data to external devices can be customized: the transport layer delegates the task of converting to a code conversion facet. To provide a stream with a suitable code conversion facet for binary output, you can insert a `double` into a file stream in a portable binary data exchange format. No such code conversion facets are provided by the Standard Library, however, and implementing such a facet is not trivial. As an alternative, you might consider implementing an entire stream buffer layer that can handle binary I/O.

- **Extending Iostreams.** In a way, you can think of iostreams as a framework that can be extended and customized. You can add input and output operators for user-defined types, or create your own formatting elements, the manipulators. You can specialize entire streams, usually in conjunction with specialized stream buffers. You can provide different locales to represent different cultural conventions, or to contain special purpose facets. You can instantiate iostreams classes for new character types, other than `char` or `wchar_t`.

2.2.4 The Internal Structure of the Iostreams Layers

As explained earlier, iostreams have two layers, one for formatting, and another for code conversion and transport of characters to and from the external device. For convenience, let's repeat here in *Figure 23* the illustration of the iostreams layers given in Figure 21 of Section 2.2.2:

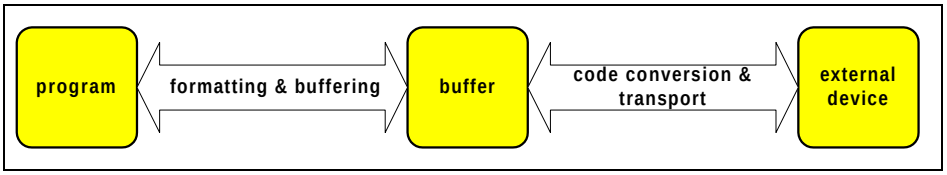


Figure 23. The iostreams layers

This section will give a more detailed description of the iostreams software architecture, including the classes and their inheritance relationship and respective responsibilities. If you would rather start using iostreams directly, go on to Section 2.3.

2.2.4.1 The Internal Structure of the Formatting Layer

Classes that belong to the formatting layer are often referred to as the *stream classes*. Figure 24 illustrates the class hierarchy of all the stream classes:

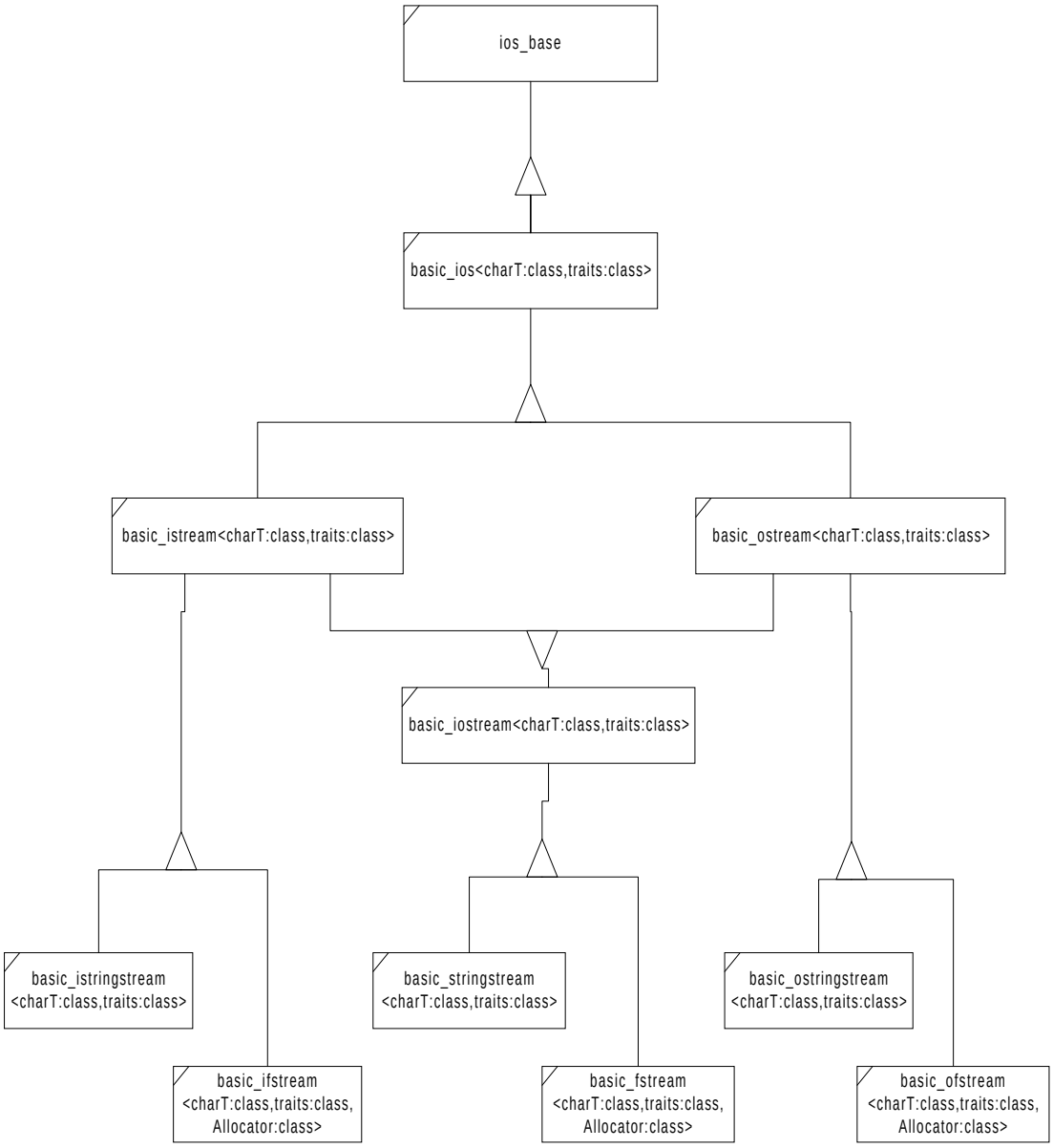


Figure 24. Internal class hierarchy of the formatting layer [14]

[14] The classes `strstream`, `istrstream`, and `ostrstream` are described in the *Class Reference*, but not this *User's Guide*. These classes are sometimes called *deprecated features* in the standard; i.e., they are provided solely for the sake of compatibility with the traditional iostreams, and will not be supported in future versions of the standard iostreams.

Let us discuss in more detail the components and characteristics of the class hierarchy given in the figure:

- **The Iostreams Base Class *ios_base*.** This class is the base class of all stream classes. Independent of character type, it encapsulates information that is needed by all streams. This information includes:
  - Control information for parsing and formatting;
  - Additional information for the user's special needs (a way to extend iostreams, as we will see later on);
  - The locale imbued on the stream;

  Additionally, ***ios_base*** defines several types that are used by all stream classes, such as format flags, status bits, open mode, exception class, etc.

- **The Iostreams Character Type-Dependent Base Class.** Here is the virtual base class for the stream classes:

  `basic_ios<class charT, class traits=char_traits<charT> >`

  - The class holds a pointer to the stream buffer, and
  - State information that reflects the integrity of the stream buffer;

  Note that ***basic_ios<>*** is a class template taking two parameters, the type of character handled by the stream, and the *character traits*.

  The type of character can be type `char` for single-byte characters, or type `wchar_t` for wide characters, or any other user-defined character type. There are instantiations for `char` and `wchar_t` provided by the Standard C++ Library.

  For convenience, there are typedefs for these instantiations:

  `typedef basic_ios<char>` **ios** and `typedef basic_ios<wchar_t>` **wios**

  Note that **ios** is not a class anymore, as it was in the traditional iostreams. If you have existing programs that use the old iostreams, they may no longer be compilable with the standard iostreams. (See list of incompatibilities in section 2.14)

- **Character Traits.** These describe the properties of a character type. Many things change with the character type, such as:
  - **The end-of-file value**. For type `char`, the end-of file value is represented by an integral constant called `EOF`. For type `wchar_t`, there is a constant defined that is called `WEOF`. For an arbitrary user-defined character type, the associated character traits define what the end-of-file value for this particular character type is.
  - **The type of the `EOF` value**. This needs to be a type that can hold the `EOF` value. For example, for single-byte characters, this type is `int`, different from the actual character type `char`.

- **The equality of two characters**. For an exotic user-defined character type, the equality of two characters might mean something different from just bit-wise equality. Here you can define it.

A complete list of character traits is given in the string section that explains character traits.

There are specializations defined for type `char` and `wchar_t`. In general, this class template is not meant to be instantiated for a character type. You should always define class template specializations.

Fortunately, the Standard C++ Library is designed to make the most common cases the easiest. The traits template parameter has a sensible default value, so usually you don't have to bother with character traits at all.

- **The Input and Output Streams.** The three stream classes for input and output are:

```
basic_istream <class charT, class traits=char_traits<charT> >
basic_ostream <class charT, class traits=char_traits<charT> >
basic_iostream<class charT, class traits=char_traits<charT> >
```

Class `istream` handles input, class `ostream` is for output. Class `iostream` deals with input *and* output; such a stream is called a *bidirectional* stream.

The three stream classes define functions for parsing and formatting, which are overloaded versions of `operator>>()` for input, called *extractors*, and overloaded versions of `operator<<()` for output, called *inserters*.

Additionally, there are member functions for unformatted input and output, like `get()`, `put()`, etc.

- **The File Streams.** The file stream classes support input and output to and from files. They are:

```
basic_ifstream<class charT, class traits=char_traits<charT> >
basic_ofstream<class charT, class traits=char_traits<charT> >
basic_fstream<class charT, class traits=char_traits<charT> >
```

There are functions for opening and closing files, similar to the C functions `fopen()` and `fclose()`. Internally they use a special kind of stream buffer, called a *file buffer*, to control the transport of characters to/from the associated file. The function of the file streams is illustrated in Figure 25:

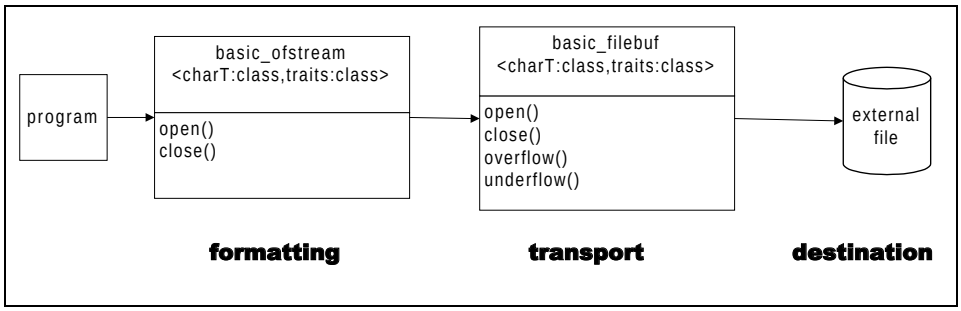


Figure 25. File I/O

- **The String Streams.** The string stream classes support in-memory I/O; that is, reading and writing to a string held in memory. They are:

  ```
  basic_istringstream<class charT, class traits=char_traits<charT> >
  basic_ostringstream<class charT, class traits=char_traits<charT> >
  basic_stringstream<class charT, class traits=char_traits<charT> >
  ```

  There are functions for getting and setting the string to be used as a buffer. Internally a specialized stream buffer is used. In this particular case, the buffer and the external device are the same. Figure 26 below illustrates how the string stream classes work:

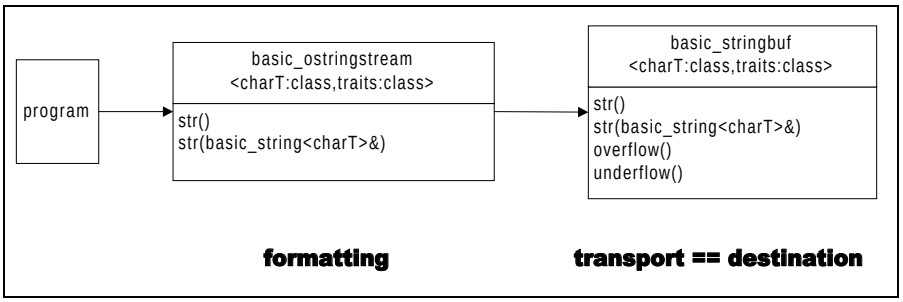
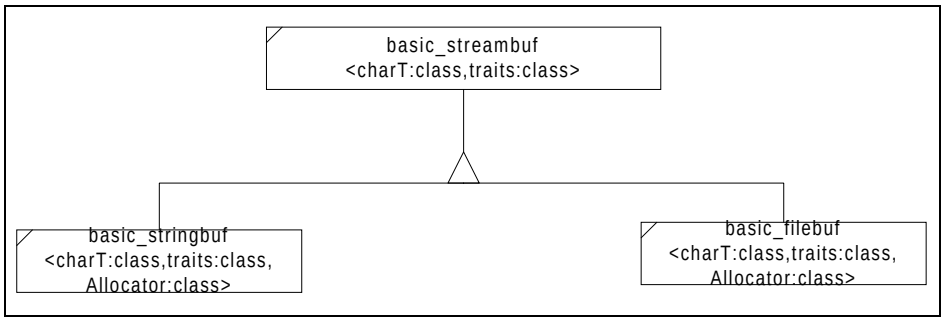


Figure 26. In-memory I/O

2.2.4.2 The Transport Layer's Internal Structure



Classes of the transport layer are often referred to as the stream buffer classes. Figure 27 gives the class hierarchy of all stream buffer classes:

Figure 27. Hierarchy of the transport layer

The stream buffer classes are responsible for transfer of characters from and to external devices.

- **The Stream Buffer.** This class represents an abstract stream buffer:

  ```
  basic_streambuf<class charT, class traits=char_traits<charT> >
  ```

  It does not have any knowledge about the external device. Instead, it defines two virtual functions, `overflow()` and `underflow()`, to perform the actual transport. These two functions have knowledge of the peculiarities of the external device they are connected to. They have to be overwritten by all concrete stream buffer classes, like file and string buffers.

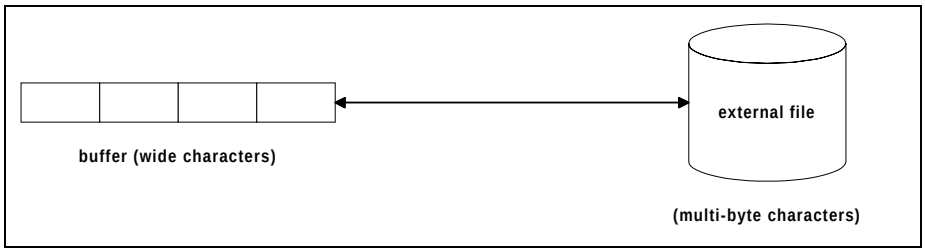
  The stream buffer class maintains two character sequences: the *get area*, which represents the input sequence read from an external device, and the *put area*, which is the output sequence to be written to the device. There are functions for providing the next character from the buffer, such as `sgetc()`, etc. They are typically called by the formatting layer in order to receive characters for parsing. Accordingly, there are also functions for placing the next character into the buffer, such as `sputc()`, etc.

  A stream buffer also carries a locale object.

- **The File Buffer.** The file buffer classes associate the input and output sequences with a file. A file buffer takes the form:

  ```
  basic_filebuf<class charT, class traits=char_traits<charT> >
  ```

  The file buffer has functions like `open()` and `close()`. The file buffer class inherits a locale object from its stream buffer base class. It uses the locale's code conversion facet for transforming the external character encoding to the encoding used internally. Figure 28 shows how the file



  buffer works:

  Figure 28. Character code conversion performed by the file buffer

- **The String Stream Buffer.** These classes implement the in-memory I/O:

  ```
  basic_stringbuf<class charT, class traits=char_traits<charT> >
  ```

With string buffers, the internal buffer and the external device are one and the same. The internal buffer is dynamic, in that it is extended if necessary to hold all the characters written to it. You can obtain copies of the internally held buffer, and you can provide a string to be copied into the internal buffer.

2.2.4.3 Collaboration of Streams and Stream Buffers

The base class ***basic_ios<>*** holds a pointer to a stream buffer. The derived stream classes, like file and string streams, contain a file or string buffer object. The stream buffer pointer of the base class refers to this embedded object. This architecture is illustrated in Figure 29 below:

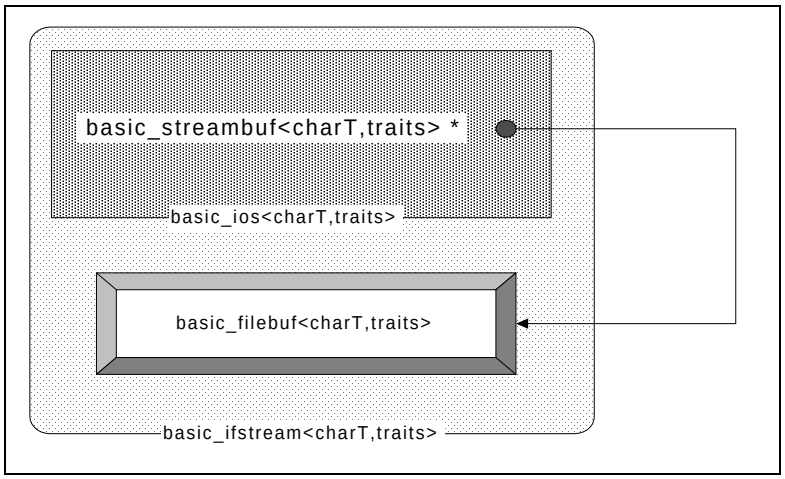


Figure 29. How an input file stream uses a file buffer

Stream buffers can be used independently of streams, as for unformatted I/O, for example. However, streams always need a stream buffer.

2.2.4.4 Collaboration of Locales and Iostreams

The base class ***ios_base*** contains a locale object. The formatting and parsing functions defined by the derived stream classes use the *numeric facets* of that locale.

The class ***basic_ios<charT>*** holds a pointer to the stream buffer. This stream buffer has a locale object, too, usually a copy of the same locale object used by the functions of the stream classes. The stream buffer's input and output functions use the *code conversion facet* of the attached locale. Figure 30 below illustrates the architecture:

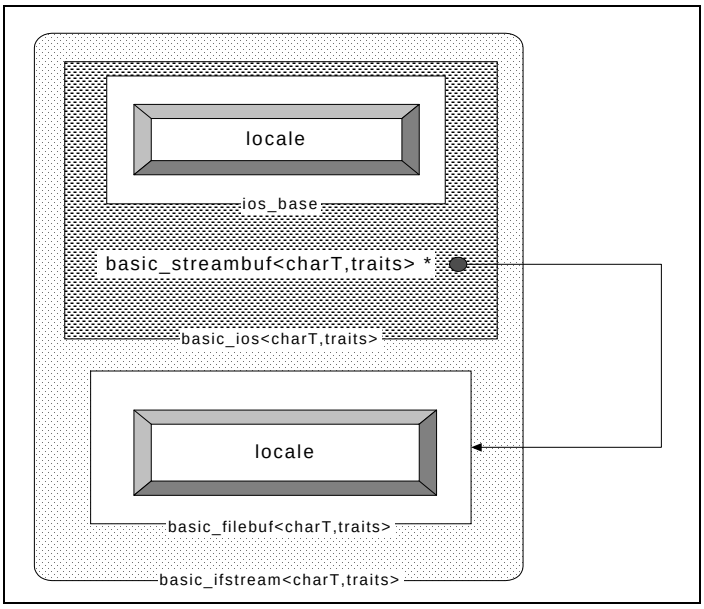


Figure 30. How an input file stream uses locales

2.3 Formatted Input/Output

This section describes the formatting facilities of iostreams. Here we begin using the predefined streams, and see how to do simple input and output. We then explore in detail how to control parsing and formatting.

2.3.1 The Predefined Streams

There are eight predefined standard streams that are automatically created and initialized at program start. These standard streams are associated with the C standard files `stdin`, `stdout`, and `stderr`, as shown in Table 3:

Table 3. Predefined standard streams with their associated C standard files

Narrow character stream	Wide character stream	Associated C standard files
`cin`	`wcin`	`stdin`
`cout`	`wcout`	`stdout`
`cerr`	`wcerr`	`stderr`
`clog`	`wclog`	`stderr`

Like the C standard files, these streams are all associated by default with the terminal.

The difference between `clog` and `cerr` is that `clog` is fully buffered, whereas output to `cerr` is written to the external device after each formatting. With a fully buffered stream, output to the actual external device is written only when the buffer is full. Thus `clog` is more efficient for redirecting output to a file, while `cerr` is mainly useful for terminal I/O. Writing to the external device after every formatting, to the terminal in the case of `cerr`, serves the purpose of synchronizing output to and input from the terminal.

The standard streams are initialized in such a way that they can be used in constructors and destructors of static objects. Also, the predefined streams are synchronized with their associated C standard files. See Section 2.10.5 for details.

2.3.2 Input and Output Operators

Now let's try to do some simple input and output to the predefined streams. The iostreams facility defines shift operators for formatted stream input and output. The output operator is the shift operator `operator<<()`, also called the *inserter* (defined in Section 2.2.4.1):

```
cout << "result: " << x << '\n';
```

Input is done through another shift operator `operator>>()`, often referred to as the *extractor* (also defined in Section 2.2.4.1):

```
cin >> x >> y;
```

Both operators are overloaded for all built-in types in C++, as well as for some of the types defined in the Standard C++ Library; for example, there are inserters and extractors for `bool`, `char`, `int`, `long`, `float`, `double`, `string`, etc. When you insert or extract a value to or from a stream, the C++ function overload resolution chooses the correct extractor operator, based on the value's type. This is what makes C++ iostreams type-safe and better than C stdio (see Section 2.2.1.1).

It is possible to print several units in one expression. For example:

```
cout << "result: " << x;
```

is equivalent to:

```
(cout.operator<<("result: ")).operator<<(x);
```

This is possible because each shift operator returns a reference to the respective stream. Almost all shift operators for built-in types are member functions of their respective stream class.[15] They are defined according to the following patterns:

```
template<class charT, class traits>
basic_istream<charT, traits>&
basic_istream<charT, traits>::operator>>(type& x)
{
  // read x
  return *this;
}
```

and:

```
template<class charT, class traits>
basic_ostream<charT, traits>&
basic_ostream<charT, traits>::operator<<(type x)
{
  // write x
  return *this;
}
```

Simple input and output of units as shown above is useful, yet not sufficient in many cases. For example, you may want to vary the way output is formatted, or input is parsed. Iostreams allow you to control the formatting features of its input and output operators in many ways. With iostreams, you can specify:

- The width of an output field and the adjustment of the output within this field;
- The precision and format of floating point numbers, and whether or not the decimal point should always be included;
- Whether you want to skip white spaces when reading from an input stream;
- Whether integral values are displayed in decimal, octal or hexadecimal format,

and many other formatting options.

---

[15] The shift operators for the character types, like `char` and `wchar_t`, are an exception to this rule; they are global functions in the standard library `namespace ::std`.

There are two mechanisms that have an impact on formatting:

- Formatting control through a stream's format state, and
- Localization through a stream's locale.

The stream's format state is the main means of format control, as we will demonstrate in the next section.

2.3.3 Format Control Using the Stream's Format State

2.3.3.1 Format Parameters

Associated with each stream are a number of *format state variables* that control the details of formatting and parsing. Format state variables are classes inherited from a stream's base class, either ***ios_base*** or ***basic_ios<charT,traits>***. There are two kinds of format parameters:

- ***Parameters that can have an arbitrary value.*** The value is stored as a private data member in one of the base classes, and set and retrieved through public member functions inherited from that base class. There are three such parameters, described in Table 4 below:

Table 4. Format parameters with arbitrary values

Access function	Defined in base class	Effect	Default
`width()`	***ios_base***	Minimal field width	0
`precision()`	***ios_base***	Precision of floating point values	6
`fill()`	***basic_ios <charT,traits>***	Fill character for padding	The space character

- ***Parameters that can have only a few different values, typically two or three.*** They are represented by one or more bits in a data member of type `fmtflags` in class ***ios_base***. These are usually called *format flags*. You can set format flags using the `setf()` function in class ***ios_base***, clear them using `unsetf()`, and retrieve them through the `flags()` function.

  Some format flags are grouped because they are mutually exclusive; for example, output within an output field can be adjusted to the left or to the right, or to an internally specified adjustment. One and only one of

the corresponding three format flags, `left`, `right`, or `internal`, can be set.[16] If you want to set one of these bits to 1, you need to set the other two to 0. To make this easier, there are *bit groups* whose main function is to reset all bits in one group. The bit group for adjustment is `adjustfield`, defined as `left | right | internal`.

Table 5 below gives an overview of all format flags and their effects on input and output operators. (For details on how the format flags affect input and output operations, see the *Class Reference* entry for ***ios_base***.) The first column below, *format flag*, lists the flag names; for example, `showpos` stands for `ios_base::showpos`. The *group* column lists the name of the group for flags that are mutually exclusive. The third column gives a brief description of the effect of setting the flag. The *stdio* column refers to format characters used by the C functions `scanf()` or `printf()` that have the same or similar effect. The last column, *default*, lists the setting that is used if you do not explicitly set the flag.

Table 5: Flags and their effects on operators

Format flag	Group	Effect	stdio	Default
	`adjustfield`	Adds fill characters to certain generated output for adjustment:		`left`[17]
`left`		left	`-`	
`right`		right	`0`	
`internal`		adds fill characters at designated internal point		
	`basefield`	Converts integer input or generates integer output in:	`%i`	`dec`
`dec`		decimal base	`%d,%u`	
`oct`		octal base	`%o`	
`hex`		hexadecimal base	`%x`	

[16] Iostreams does not prevent you from setting other invalid combinations of these flags, however.

[17] Initially, none of the bits is set. This is more or less equivalent to `left`.

`fixed`
`scientific`
	`floatfield`	Generates floating point output:
		in fixed-point notation
		in scientific notation
			`%g,%G`
			`%f`
			`%e,%E`
				`fixed`

`boolalpha`		Inserts and extracts `bool` values in alphabetic format		0
`showpos`		Generates a `+` sign in non-negative generated numeric output	`+`	0
`showpoint`		Always generates a decimal-point in generated floating-point output	`#`	0
`showbase`		Generates a prefix indicating the numeric base of a generated integer output		0
`skipws`		Skips leading white space before certain input operations		1
`unitbuf`		Flushes output after each formatting operation		0

`uppercase`		Replaces certain lowercase letters with their uppercase equivalents in generated output	`%X`
			`%E`
			`%G`
				0

The effect of setting a format parameter is usually permanent; that is, the parameter setting is in effect until the setting is explicitly changed. The only exception to this rule is the field width. The width is automatically reset to its default value `0` after each input or output operation that uses the field width.[18] Here is an example:

[18] The details of exactly when width is reset to zero are not specified in the standard at this time.

```
int i; char* s[11];
cin >> setw(10) >> i >> s;                                    \\1
cout << setw(10) <<i << s;                                   \\2
```

`//1` Extracting an integer is independent of the specified field width. The extractor for integers always reads as many digits as belong to the integer. As extraction of integers does not use the field width setting, the field width of `10` is still in effect when a character sequence is subsequently extracted. Only 10 characters will be extracted in this case. After the extraction, the field width is reset to `0`.

`//2` The inserter for integers uses the specified field width and fills the field with padding characters if necessary. After the insertion, it resets the field width to `0`. Hence, the subsequent insertion of the string will not fill the field with padding characters for a string with less than 10 characters.

Please note: **With the exception of the field width, all format parameter settings are permanent. The field width parameter is reset after each use.**

The following code sample shows how you can control formatting by using some of the parameters:

```
#include <iostream>
using namespace ::std;
// …
ios_base::fmtflags original_flags = cout.flags();            \\1
cout<< 812<<'|';
cout.setf(ios_base::left,ios_base::adjustfield);             \\2
cout.width(10);                                              \\3
cout<< 813 << 815 << '\n';
cout.unsetf(ios_base::adjustfield);                          \\4
cout.precision(2);
cout.setf(ios_base::uppercase|ios_base::scientific);         \\5
cout << 831.0 << ' ' << 8e2;
cout.flags(original_flags);                                  \\6
```

`//1` Store the current format flag setting, in order to restore it later on.

`//2` Change the adjustment from the default setting `right` to `left`.

`//3` Set the field width from its default `0` to `10`. A field width of `0` means that no padding characters are inserted, and this is the default behavior of all insertions.

`//4` Clear the adjustment flags.

`//5` Change the precision for floating-point values from its default `6` to `2`, and set yet another couple of format flags that affect floating-point values.

`//6` Restore the original flags.

The output is:

```
812|813       815
8.31E+02 8.00E+02
```

2.3.3.2 Manipulators

Format control requires calling a stream's member functions. Each such call interrupts the respective shift expression. But what if you need to change formats within a shift expression? This is possible in iostreams. Instead of writing:

```
cout<< 812 << '|';
cout.setf(ios_base::left,ios_base::adjustfield);
cout.width(10);
cout<< 813 << 815 << '\n';
```

you can write:

```
cout<< 812 << '|' << left << setw(10) << 813 << 815 << endl;
```

In this example, objects like `left`, `setw`, and `endl` are called *manipulators*. A manipulator is an object of a certain type; let's call the type `manip` for the time being. There are overloaded versions of `basic_istream <charT,traits>:: operator>>()` and `basic_ostream <charT,traits>:: operator<<()` for type `manip`. Hence a manipulator can be extracted from or inserted into a stream together with other objects that have the shift operators defined. (Section 2.8 explains in greater detail how manipulators work and how you can implement your own manipulators.)

The effect of a manipulator need not be an actual input to or output from the stream. Most manipulators set just one of the above described format flags, or do some other kind of stream manipulation. For example, an expression like:

```
cout << left;
```

is equivalent to:

```
cout.setf (ios_base::left, ios_base::adjustfield);.
```

Nothing is inserted into the stream. The only effect is that the format flag for adjusting the output to the left is set.

On the other hand, the manipulator `endl` inserts the newline character to the stream, and flushes to the underlying stream buffer. The expression:

```
cout << endl;
```

is equivalent to:

```
cout << '\n'; cout.flush();
```

Some manipulators take arguments, like `setw(int)`. The `setw` manipulator sets the field width. The expression:

```
cout << setw(10);
```

is equivalent to:

```
cout.width(10);
```

In general, you can think of a manipulator as an object you can insert into or extract from a stream, in order to manipulate that stream.

Some manipulators can be applied only to output streams, others only to input streams. Most manipulators change format bits only in one of the stream base classes, ***ios_base*** or ***basic_ios<charT,traits>***. These can be applied to input and output streams.

Table 6 below gives an overview of all manipulators defined by iostreams. The first column, **Manipulator**, lists its name. All manipulators are classes defined in the namespace `::std`. The second column, **Use**, indicates whether the manipulator is intended to be used with istreams (`i`), ostreams (`o`), or both (`io`). The third column, **Effect**, summarizes the effect of the manipulator. The last column, **Equivalent**, lists the corresponding call to the stream's member function.

Note that the second column indicates only the *intended* use of a manipulator. In many cases, it is possible to apply an output manipulator to an input stream, and vice versa. Generally, this kind of non-intended manipulation is harmless in that it has no effect. For instance, if you apply the output manipulator `showpoint` to an input stream, the manipulation will simply be ignored. However, if you use an output manipulator on a bidirectional stream during input, the manipulation will affect not current input operations, but subsequent output operations.

Table 6: Manipulators

Manipulator	Use	Effect	Equivalent
`boolalpha`	`io`	Puts `bool` values in alphabetic format	`io.setf(ios_base::boolalpha)`
`dec`	`io`	Converts integers to/from decimal notation	`io.setf(ios_base::dec, ios_base::basefield)`
`endl`	`o`	Inserts newline and flushes buffer	`o.put(o.widen('\n'));` `o.flush()`
`ends`	`o`	Inserts end of string character	`o.put(o.widen('\0'))`
`fixed`	`o`	Puts floating point values in fixed-point notation	`o.setf(ios_base::fixed, ios_base::floatfield)`
`flush`	`o`	Flushes stream buffer	`o.flush()`

`hex`	`io`	Converts integers to/from hexadecimal notation	`io.setf(ios_base::hex, ios_base::basefield)`
`internal`	`o`	Adds fill characters at a designated internal point	`o.setf(ios_base::internal, ios_base::adjustfield)`
`left`	`o`	Adds fill characters for adjustment to the left	`o.setf(ios_base::left, ios_base::adjustfield)`
`noboolalpha`	`io`	Resets the above	`io.unsetf(ios_base::boolalpha)`
`noshowbase`	`o`	Resets the above	`o.unsetf (ios_base::showbase)`
`noshowpoint`	`o`	Resets the above	`o.unsetf (ios_base::showpoint)`
`noshowpos`	`o`	Resets the above	`o.unsetf (ios_base::showpos)`
`noskipws`	`i`	Resets the above	`i.unsetf(ios_base::skipws)`
`nounitbuf`	`o`	Resets the above	`o.unsetf(ios_base::unitbuf)`
`nouppercase`		Resets the above	`o.unsetf (ios_base::uppercase)`
`oct`	`io`	Converts to/from octal notation	`io.setf(ios_base::oct, ios_base::basefield)`
`resetiosflags (ios_base::fmtflags mask)`	`io`	Clears ***ios*** flags	`io.setf((ios_base::fmtflags)0, mask)`
`right`	`o`	Adds fill characters for adjustment to the right	`o.setf(ios_base::right, ios_base::adjustfield)`
`scientific`		Puts floating point values in scientific notation	`o.setf(ios_base::scientific, ios_base::floatfield)`
`setbase (int base)`	`io`	Sets base for integer notation (base = 8, 10, 16)	`io.setf (base == 8?ios_base::oct: base == 10 ? ios_base::dec : base == 16 ? ios_base::hex : ios_base::fmtflags(0), ios_base::basefield)`
`setfill(charT c)`	`io`	Sets fill character for padding	`io.fill(c)`

setiosflags (ios_base::fmtflags mask)	io	Sets ***ios*** flags	io.setf(mask)
setprecision (int n)	io	Sets precision of floating point values	io.precision(n)
setw(int n)	io	Sets minimal field width	io.width(n)
showbase	o	Generates a prefix indicating the numeric base of an integer	o.setf(ios_base::showbase)
showpoint	o	Always generates a decimal-point for floating-point values	o.setf(ios_base::showpoint)
showpos	o	Generates a + sign for non-negative numeric values	o.setf(ios_base::showpos)
skipws	i	Skips leading white space	i.setf(ios_base::skipws)
unitbuf	o	Flushes output after each formatting operation	o.setf(ios_base::unitbuf)
uppercase	o	Replaces certain lowercase letters with their uppercase equivalents	o.setf(ios_base::uppercase)
ws	i	Skips white spaces	

2.3.4 Localization Using the Stream's Locale

Associated with each stream is a locale that impacts the parsing and formatting of numeric values. This is how localization of software takes place. As discussed in the section on locale, the representation of numbers often depends on cultural conventions. In particular, the *decimal point* need not be a period, as in the following example:

```
cout.imbue(locale("De_DE"));
cout << 1000000.50 << endl;
```

The output will be:

```
1000000,50
```

Other cultural conventions, like the grouping of digits, are irrelevant. There is no formatting of numeric values that involves grouping.[19]

2.3.5 Formatted Input

In principle, input and output operators behave symmetrically. There is only one important difference: for output you control the precise format of the inserted character sequence, while for input the format of an extracted character sequence is never exactly described.

This is for practical reasons. You may want to extract the next floating point value from a stream, for example, without necessarily knowing its exact format. You want it whether it is signed or not, or in exponential notation with a small or capital E for the exponent, etc. Hence, extractors in general accept an item in any format permitted for its type.

Formatted input is handled as follows:

1. Extractors automatically ignore all white space characters (blanks, tabulators, newlines [20]) that precede the item to be extracted.
2. When the first relevant character is found, they extract characters from the input stream until they find a separator; that is, a character that does not belong to the item. White space characters in particular are separators.
3. The separator remains in the input stream and becomes the first character extracted in a subsequent extraction.

Several format parameters, which control insertion, are irrelevant for extraction. The format parameter fill character, `fill()`, and the adjustment flags, `left`, `right`, and `internal`, have no effect on extraction. The field width is relevant only for extraction of strings, and ignored otherwise.

2.3.5.1 Skipping Characters

---

[19] The standard does not specify whether the grouping information, that is contained in a stream's locale's `numpunct` facet, is ignored or taken into account if present. In any case, there are no manipulators that allow to switch on and off the grouping.

[20] The classification of a character as a *white space character* depends on the character set used. The extractor takes the information from the locale's ctype facet.

You can use the manipulator `noskipws` to switch off the automatic skipping of white space characters. For example, extracting white space characters may be necessary if you expect the input has a certain format, and you need to check for violations of the format requirements. This procedure is shown in the following code:

```
cin >> noskipws;
char c;
do
{ float fl;
  c = ' '; cin >> fl >> c;  // extract number and separator
if (c == ',' || c == '\n')  // next char is ',' or newline ?
    process(fl);            // yes: use the number
}
while (c == ',');
if (c != '\n') error();     // no: error!
```

If you have to skip a sequence of characters other than white spaces, you can use the istream's member function `ignore()`. The call:

```
basic_ifstream<myChar,myTraits> InputStream("file-name");
InputStream.ignore(numeric_limits<streamsize>::max()
                   ,myChar('\n'));
```

or, for ordinary tiny characters of type `char`:

```
ifstream InputStream("file-name");
InputStream.ignore(INT_MAX,'\n');
```

ignores all characters until the end of the line. This example uses a file stream that is not predefined. File streams are described in Section 2.5.3.

2.3.5.2 Input of Strings

When you extract strings or character arrays from an input stream, characters are read until:

- A white space character is found, or
- The end of the input is reached, or
- A certain number of characters are extracted, if `width() != 0`: In case of a string this number is the field width `width()`. In case of a character array this number is `width()-1`.

Note that the field width will be reset to `0` after the extraction of a string.

There are subtle differences between extracting a character sequence into a character array and extracting it into a string object. For example:

```
char buf[SZ];
cin >> buf;
```

is different from:

```
string s;
cin >> s;
```

Extraction into a string is safe, because strings automatically extend their capacity as necessary. You can extract as many characters as you want since the string always adjusts its size accordingly. Character arrays, on the other hand, have fixed size and cannot dynamically extend their capacity. If you extract more characters than the character array can take, the extractor writes beyond the end of the array. To prevent this, you must set the field width as follows each time you extract characters into a character array:

```
char buf[SZ];
cin >> width(SZ) >> buf;
```

2.4 Error State of Streams

It probably comes as no surprise that streams have an error state. Our examples have avoided it to this point, so we'll deal with it now. When an error occurs, flags are set in the state according to the general category of the error. Flags and their error categories are summarized in Table 7 below:

Table 7: Flags and corresponding error categories

iostate flag	Error category
`ios_base::goodbit`	Everything's fine
`ios_base::eofbit`	An input operation reached the end of an input sequence
`ios_base::failbit`	An input operation failed to read the expected character, or An output operation failed to generate the desired characters
`ios_base::badbit`	Indicates the loss of integrity of the underlying input or output sequence

Note that the flag `ios_base::goodbit` is not really a flag; its value, zero, indicates the absence of any error flag. It means the stream is OK. By convention, all input and output operations have no effect once the stream state is different than zero.

There are several situations when both `eofbit` and `failbit` are set; however, the two have different meanings and do not always occur in conjunction. The flag `ios_base::eofbit` is set when there is an attempt to read past the end of an input sequence. This occurs in the following two typical examples:

1. Assume the extraction happens character-wise. Once the last character is read, the stream is still in good state; `eofbit` is not yet set. Any subsequent extraction, however, will be considered an attempt to read past the end of the input sequence. Thus, `eofbit` will be set.

2. If you do not read character-wise, but extract an integer or a string, for example, you will always read past the end of the input sequence. This is because the input operators read characters until they find a separator, or hit the end of the input sequence. If the input contains the sequence `... 912749<eof>` and an integer is extracted, `eofbit` will be set.

The flag `ios_base::failbit` is set as the result of a read or write operation that fails. For example, if you try to extract an integer from an input sequence containing only white spaces, the extraction fails, and the `failbit` is set. Let's see whether `failbit` would be set in the previous examples:

1. After reading the last available character, the extraction not only reads past the end of the input sequence; it also fails to extract the requested character. Hence, `failbit` is set in addition to `eofbit`.

2. Here it is different. Although the end of the input sequence is reached by extracting the integer, the input operation does not fail and the desired integer will indeed be read. Hence, in this situation only the `eofbit` will be set.

In addition to these input and output operations, there are other situations that can trigger failure. For example, file streams set `failbit` if the associated file cannot be opened (see Section 2.5).

The flag `ios_base::badbit` indicates problems with the underlying stream buffer. These problems could be:

- ***Memory shortage***. There is no memory available to create the buffer, or the buffer has size zero for other reasons[21], or the stream cannot allocate memory for its own internal data[22], as with `iword` and `pword`.

- ***The underlying stream buffer throws an exception***. The stream buffer might lose its integrity, as in memory shortage, or code conversion failure, or an unrecoverable read error from the external device. The stream buffer can indicate this loss of integrity by throwing an exception, which is caught by the stream and results in setting the `badbit` in the stream's state.

Generally, you should keep in mind that `badbit` indicates an error situation that is likely to be unrecoverable, whereas `failbit` indicates a situation that

---

[21] The stream buffer can be created as the stream's responsibility, or the buffer can be provided from outside the stream, so inadvertently the buffer could have zero size.

[22] The standard does not yet specify whether the inability to allocate memory for the stream's internal data will result in a `badbit` set or a `bad_alloc` or `ios_base::failure` thrown.

might allow you to retry the failed operation. The flag `eofbit` simply indicates the end of the input sequence.

What can you do to check for such errors? You have two possibilities for detecting stream errors:

- You can declare that you want to have an exception raised once an error occurs in any input or output operation, or
- You can actively check the stream state after each input or output operation.

We will explore these possibilities in the next two sections.

2.4.1 Checking the Stream State

Let's first examine how you can check for errors using the stream state. A stream has several member functions for this purpose, which are summarized with their effects in the Table 8:

Table 8. Stream member functions for error checking

***ios_base* member function**	**Effect**
`bool good()`	True if no error flag is set
`bool eof()`	True if `eofbit` is set
`bool fail()`	True if `failbit` or `badbit` is set
`bool bad()`	True if `badbit` is set
`bool operator!()`	As `fail()`
`operator void*()`	Null pointer if `fail()` and non-null value otherwise
`iostate rdstate()`	Value of stream state

It is a good idea to check the stream state in some central place, for example:

```
if (!cout) error();
```

The state of `cout` is examined with `operator!()`, which will return `true` if the stream state indicates an error has occurred.

An ostream can also appear in a boolean position to be tested as follows:

```
if (cout << x) // okay!
```

The magic here is the `operator void*()` that returns a non-zero value when the stream state is non-zero.

Finally, the explicit member functions can also be used:

```
if (cout << x, cout.good()) // okay!;
```

Note that there is a difference between `good()` and `operator!()`. The function `good()` takes all flags into account; `operator!()` and `fail()` ignore `eofbit`.

2.4.2 Catching Exceptions

By default a stream does not throw any exceptions.[23] You have to explicitly activate an exception because a stream contains *an exception mask*. Each flag in this mask corresponds to one of the error flags. For example, once the `badbit` flag is set in the exception mask, an exception will be thrown each time the `badbit` flag gets set in the stream state. The following code demonstrates how to activate an exception on an input stream `InStr`:

```
try {
   InStr.exceptions(ios_base::badbit | ios_base::failbit);    \\1
   in >> x;
   // do lots of other stream i/o
}
catch(ios_base::failure& exc)                                 \\2
{   cerr << exc.what() << endl;
    throw;
}
```

`//1` In calling the `exceptions()` function, you indicate what flags in the stream's state shall cause an exception to be thrown.[24]

---

[23] Actually, the standard does not yet specify how memory allocation errors are to be handled in iostreams. Basically, the two models are:

- A `bad_alloc` exception is thrown, regardless of whether or not any bits in the exception mask are set.

- The streams layer catches `bad_alloc` exceptions thrown during allocation of its internal resources, `iword` and `pword`. It would then set `badbit` or `failbit`. An exception would be thrown only if the respective bit in the exception mask asks for it. It must be specified whether the exception thrown in such a case would be `ios_failure` or the original `bad_alloc`.

Moreover, the streams layer must catch all exceptions thrown by the stream buffer layer. It sets `badbit` and rethrows the original exception if the exception mask permits it.

[24] Note that each change of either the stream state or the exception mask can result in an exception thrown. This is because

`//2` Objects thrown by the stream's operations are of types derived from `ios_base::failure`. Hence this catch clause will catch all stream exceptions in principle. We qualify this generalization because a stream might fail to catch certain exceptions, e.g., `bad_alloc`, so that exceptions other than `ios_base::failure` might be raised. That's how exception handling in C+ works: you never know what exceptions will be raised.

Generally, it is a good idea to activate the `badbit` exception and suppress the `eofbit` and `failbit` exceptions, because the latter do not represent exceptional states. A `badbit` situation, however, is likely to be a serious error condition similar to the memory shortage indicated by a `bad_alloc` exception. Unless you want to suppress exceptions thrown by iostreams altogether, we would recommend that you switch on the `badbit` exception and turn off `eofbit` and `failbit`.

2.5 File Input/Output

File streams allow input and output to files. Unlike the C stdio functions for file I/O, however, file streams follow Stroustrup's idiom: "Resource acquisition is initialization."[25] In other words, file streams provide an advantage in that you can open a file on construction of a stream, and the file will be closed automatically on destruction of the stream. Consider the following code:

```
void use_file(const char* fileName)
{
   FILE* f = fopen("fileName", "w");
   // use file
   fclose(f);
}
```

If an exception is thrown while the file is in use here, the file will never be closed. With a file stream, however, the file will be closed whenever the file stream goes out of scope, as in the following example:

```
void use_file(const char* fileName)
{
   ofstream f("fileName");
   // use file
}
```

Here the file will be closed even if an exception occurs during use of the open file.

There are three class templates that implement file streams: ***basic_ifstream <charT,traits>***, ***basic_ofstream <charT,traits>***, and ***basic_fstream***

---

the functions `steatite()` and `exception()` raise an exception in case the exception mask requires it.

[25] See Bjarne Stroustrup, *The C++ Programming Language*, p.308ff.

<charT,traits>. These templates are derived from the stream base class ***basic_ios <charT, traits>***. Therefore, they inherit all the functions for formatted input and output described in Section 2.3, as well as the stream state. They also have functions for opening and closing files, and a constructor that allows opening a file and connecting it to the stream. For convenience, there are the regular typedefs `ifstream`, `ofstream`, and `fstream`, with `wifstream`, `wofstream`, and `wfstream` for the respective tiny and wide character file streams.

The buffering is done through a specialized stream buffer class, ***basic_filebuf <charT,traits>***.

2.5.1 The Difference between Predefined File Streams (cin, cout, cerr, and clog) and File Streams

The main differences between a predefined standard stream and a file stream are:

- A file stream needs to be connected to a file before it can be used. The predefined streams can be used right away, even in static constructors that are executed before the `main()` function is called.
- You can reposition a file stream to arbitrary file positions. This usually does not make any sense with the predefined streams, as they are connected to the terminal by default.

2.5.2 Code Conversion in Wide Character Streams

In a large character set environment, a file is assumed to contain multibyte characters. To provide the contents of a such a file as a wide character sequence for internal processing, `wifstream` and `wofstream` perform corresponding conversions. The actual conversion is delegated to the file buffer, which relays the task to the imbued locale's code conversion facet.

2.5.3 File Streams

2.5.3.1 Creating and Opening File Stream Objects

There are two ways to create a file stream[26]: you can create an empty file stream, open a file, and connect it to the stream later on; or you can open the file and connect it to a stream at construction time. These two procedures are demonstrated in the two following examples, respectively:

```
ifstream file;                                          \\1
…;
file.open(argv[1]);                                     \\2
if (!file) // error: unable to open file for input
```

or:

```
ifstream source("src.cpp");                             \\3
if (!source) // error: unable to open src.cpp for input
```

`//1` A file stream is created that is not connected to any file. Any operation on the file stream will fail.

`//2` Here a file is opened and connected to the file stream. If the file cannot be opened, `ios_base::failbit` will be set; otherwise, the file stream is now ready for use.

`//3` Here the file is both opened and connected to the stream.

2.5.3.2 Checking a File Stream's Status

Generally you can check whether the attempt to open a file was successful by examining the stream state afterwards; `failbit` will be set in case of failure.

There is also a function called `is_open()`that indicates whether a file stream is connected to an open file. This function does *not* mean that a previous call to `open()` was successful. To understand the subtle difference, consider the case of a file stream that is already connected to a file. Any subsequent call to `open()` will fail, but `is_open()` will still return true, as shown in the following code:

---

[26] The traditional iostreams supported a constructor, taking a file descriptor, that allowed connection of a file stream to an already open file. This is not available in the standard iostreams. However, Rogue Wave's implementation of the standard iostreams provides a corresponding extension (see section 2.15.1 for reference).

```
void main(int argc, char* argv[])
{
  if (argc > 2)
  {  ofstream fil;                                          //1
     fil.open(argv[1]);
     // ...
     fil.open(argv[2]);                                     //2
     if (fil.fail())                                        //3
     {  // open failed  }
     if (fil.is_open())                                     //4
     {  // connected to an open file  }
}
```

`//1` Open a file and connect the file stream to it.

`//2` Any subsequent open on this stream will fail.

`//3` Hence the `failbit` will be set.

`//4` However, `is_open()` still returns true, because the file stream still is connected to an open file.

2.5.3.3 Closing a File Stream

In the example above, it would be advisable to close the file stream before you try to connect it to another file. This is done implicitly by the file streams destructor in the following:

```
void main(int argc, char* argv[])
{
  if (argc > 2)
  {  ofstream fil;
     fil.open(argv[1]);
     // ...
  }                                                         //1
  {  ofstream fil;
     fil.open(argv[2]);
    // ...
  }
}
```

`//1` Here the file stream `fil` goes out of scope and the file it is connected to will be closed automatically.

You can explicitly close the connected file. The file stream is then empty, until it is reconnected to another file:

```
ifstream f;                                                 //1
for (int i=1; i<argc; ++i)
{
    f.open(argv[i]);                                        //2
    if (f)                                                  //3
    {
        process(f);                                         //4
        f.close();                                          //5
    }
    else
        cerr << "file " << argv[i] << " cannot be opened.\n";
}
```

`//1` An empty file stream is created.

`//2` A file is opened and connected to the file stream.

`//3` Here we check whether the file was successfully opened. If the file could not be opened, the `failbit` would be set.

`//4` Now the file stream is usable, and the file's content can be read and processed.

`//5` Close the file again. The file stream is empty again.

2.5.4 The Open Mode

There may be times when you want to modify the way in which a file is opened or used in a program. For example, in some cases it is desirable that writes append to the end of a file rather than overwriting the existing values. The file stream constructor takes a second argument, the *open mode*, that allows such variations to be specified. Here is an example:

```
fstream Str("inout.txt",
            ios_base::in|ios_base::out|ios_base::app);
```

2.5.4.1 The Open Mode Flags

The open mode argument is of type `ios_base::openmode`, which is a bitmask type like the format flags and the stream state. The following bits are defined in Table 9:

Table 9. Flag names and effects

Flag Names	Effects
`ios_base::in`	Open file for reading
`ios_base::out`	Open file for writing
`ios_base::ate`	Start position is at file end
`ios_base::app`	Append file; i.e., always writes to the end of the file
`ios_base::trunc`	Truncate file; i.e., delete file content
`ios_base::binary`	Binary mode

2.5.4.1.1 The `in` and `out` Open Modes

Input (and output) file streams always have the `in` (or `out`) open mode flag set implicitly. An output file stream, for instance, knows that it is in output mode and you do not have to set the output mode explicitly. Instead of writing:

```
ofstream Str("out.txt",ios_base::out|ios_base::app);
```

you can simply say:

```
ofstream Str("out.txt",ios_base::app);
```

Bidirectional file streams, on the other hand, do not have the flag set implicitly. This is because a bidirectional stream does not have to be in both input and output mode in all cases. You might want to open a bidirectional stream for reading only or writing only. Bidirectional file streams therefore have no implicit input or output mode. You always have to set a bidirectional file stream's open mode explicitly.

2.5.4.1.2 The Open modes `ate`, `app`, and `trunc`

Each file maintains a *file position* that indicates the position in the file where the next byte will be read or written. When a file is opened, the initial file position is usually at the beginning of the file. The open modes `ate` (meaning *at end*) and `app` (meaning *append*) change this default to the end of the file.

There is a subtle difference between `ate` and `app` mode. If the file is opened in append mode, all output to the file is done at the current end of the file, regardless of intervening repositioning. Even if you modify the file position to a position before the file's end, you cannot write there. With at-end mode, you can navigate to a position before the end of file and write to it.

If you open an already existing file for writing, you usually want to overwrite the content of the output file. The open mode `trunc` (meaning *truncate*) has the effect of discarding the file content, in which case the initial file length is set to zero. Therefore, if you want to replace a file's content rather than extend the file, you have to open the file in `out|trunc`.[27] Note that the file position will be at the beginning of the file in this case, which is exactly what you expect for overwriting an output file.

If you want to extend an output file, you open it in at-end or append mode. In this case, the file content is retained because the `trunc` flag is not set, and the initial file position is at the file's end. However, you may additionally set the `trunc` flag; the file content will be discarded and the output will be done at the end of an empty file.

---

[27] . For output file streams the open mode `out` is equivalent to `out|trunc`, i.e. you can omit the `trunc` flag. For bidirectional file streams, however, `trunc` must always be explicitly specified.

Input mode only works for files that already exist. Otherwise, the stream construction will fail, as indicated by `failbit` set in the stream state. Files that are opened for writing will be created if they do not yet exist. The constructor only fails if the file cannot be created.

2.5.4.1.3 The `binary` Open Mode

The `binary` open mode is explained in section 2.5.4.3.

2.5.4.2 Combining Open Modes

The effect of combining these open modes is similar to the mode argument of the C library function `fopen(name,mode)`. Table 10 gives an overview of all permitted combinations of open modes for text files and their counterparts in C stdio. Combinations of modes that are not listed in the table (such as both `trunc` and `app`) are invalid, and the attempted `open()` operation will fail.

Table 10: Open modes and their C stdio counterparts

Open Mode	C stdio Equivalent	Effect
`in`	`"r"`	Open text file for reading only
`out|trunc` `out`	`"w"`	Truncate to zero length, if existent, or create text file for writing only
`out|app`	`"a"`	Append; open or create text file only for writing at end of file
`in|out`	`"r+"`	Open text file for update (reading and writing)
`in|out|trunc`	`"w+"`	Truncate to zero length, if existent, or create text file for update
`in|out|app`	`"a+"`	Append; open or create text file for update, writing at end of file

2.5.4.3 Default Open Modes

The open mode parameter in constructors and open() functions of file stream classes have a default value. The default open modes are listed in Table 11. Note that abbreviations are used; e.g., `ifstream` stands for `basic_ifstream <charT,traits>`.

Table 11. Default open modes

File Stream	Default Open Mode
`ifstream`	`in`
`ofstream`	`out`

fstream	in|out

2.5.5 Binary and Text Mode

The representation of text files varies among operating systems. For example, the end of a line in a UNIX environment is represented by the *linefeed* character `'\n'`. On PC-based systems, the end of the line consists of two characters, *carriage return* `'\r'` and *linefeed* `'\n'`. The end of the file differs as well on these two operating systems. Peculiarities on other operating systems are also conceivable.

To make programs more portable among operating systems, an automatic conversion can be done on input and output. The carriage return or linefeed sequence, for example, can be converted to a single `'\n'` character on input; the `'\n'` can be expanded to `"\r\n"` on output. This conversion mode is called *text mode*, as opposed to *binary mode*. In binary mode, no such conversions are performed.

The mode flag `ios_base::binary` has the effect of opening a file in binary mode. This has the effect described above; in other words, all automatic conversions, such as converting `"\r\n"` to `'\n'`, will be suppressed. [28]

If you have to process a binary file you should always set the `binary` mode flag, because most likely you will not want any kind of implicit, system-specific conversion being performed.

The effect of the binary open mode is frequently misunderstood. It does *not* put the inserters and extractors into a binary mode, and hence suppress the formatting they usually perform. Binary input and output is done solely by `basic_istream <charT> ::read()` and `basic_ostream <charT> ::write()`.

2.6 In-Memory Input/Output

The iostreams facility supports not only input and output to external devices like files. It also allows in-memory parsing and formatting. Source and sink of the characters read or written becomes a string held somewhere in memory. You use in-memory I/O if the information to be read is already available in the form of a string, or if the formatted result will be processed as a string. For example, to interpret the contents of the string `argv[1]` as an integer value, the code might look like this:

```
int i;
if (istringstream(argv[1]) >> i)                    //1
```

---

[28] Basically the `binary` mode flag is passed on to the respective operating system's service function, which means that in principle *all* system-specific conversions will be suppressed, not only the carriage return / linefeed handling.

```
    // use the value of i
```

`//1` The parameter of the input string stream constructor is a string; here a character array, namely `argv[1]`, is provided as an argument and is implicitly converted to a string. From this newly constructed input string stream, which contains `argv[1],` an integer value is extracted.

The inverse operation, taking a value and converting it to characters that are stored in a string, might look like this:

```
struct date {
  int day,month,year;
} today = {8,4,1996};
ostringstream ostr;                                                    //1
ostr << today.month << '-' << today.day <<'-' << today.year;  //2
if (ostr)
   display(ostr.str());                                                //3
```

`//1` An output string stream is allocated.

`//2` Values are inserted into the output string stream.

`//3` The result of the formatting can be retrieved in the form of a string, which is returned by `str()`.

As with file streams, there are three class templates that implement string streams: ***basic_istringstream <charT,traits,Allocator>***, ***basic_ostringstream <charT,traits,Allocator>***, and ***basic_stringstream <charT,traits,Allocator>***. These are derived from the stream base classes, ***basic_istream <charT, traits>***, ***basic_ostream <charT, traits>***, and ***basic_iostream <charT, traits>***. Therefore they inherit all the functions for formatted input and output described in Section 2.3, as well as the stream state. They also have functions for setting and retrieving the string that serves as source or sink, and constructors that allow you to set the string before construction time. For convenience, there are the regular typedefs `istringstream`, `ostringstream`, and `stringstream,` with `wistringstream`, `wostringstream`, and `wstringstream` for the respective tiny and wide character string streams.

The buffering is done through a specialized stream buffer class, ***basic_stringbuf <charT,traits,Allocator>***.

2.6.1 The Internal Buffer

String streams can take a string, provided either as an argument to the constructor, or set later through the `str(const basic_string <charT, traits, Allocator>&)` function. This string is copied into an internal buffer, and serves as source or sink of characters to subsequent insertions or extractions. Each time the string is retrieved through the `str()` function, a copy of the internal buffer is created and returned.

Output string streams are *dynamic*.[29] The internal buffer is allocated once an output string stream is constructed. The buffer is automatically extended during insertion each time the internal buffer is full.

Input string streams are always *static*. You can extract as many items as are available in the string you provided the string stream.

2.6.2 The Open Modes

The only open modes that have an effect on string streams are `in`, `out`, `ate`, and `app`. They have more or less the same meaning that they have with file streams (see section 2.5.4).

The `binary` open mode is irrelevant, because there is no conversion to and from the dependent file format of the operating system. The `trunc` open mode is simply ignored.

2.7 Input/Output of User-Defined Types

One of the major advantages of the iostreams facility is extensibility. Just as you have inserters and extractors for almost all types defined by the C++ language and library, you can implement extractors and inserters for all your own user-defined types. To avoid surprises, the input and output of user-defined types should follow the same conventions used for insertion and extraction of built-in types. In this section, you will find guidelines for building a typical extractor and inserter for a user-defined type.

2.7.1 An Example Using a User-Defined Type

Let us work through a complete example with the following date class as the user-defined type:

```
class date {
public:
   date(int d, int m, int y);
   date(const tm& t);
   date();
   // more constructors and useful member functions
private:
   tm tm_date;
};
```

This class has private data members of type `tm`, which is the time structure defined in the C library (in header file `<ctime>`).

---

[29] This was different in the old iostreams, where you could have dynamic and static output streams. See section 2.14.4 for further details.

2.7.2 A Simple Extractor and Inserter for the Example

Following the read and write conventions of iostreams, we would insert and extract the date object like this:

```
date birthday(2,6,1952);
cout << birthday << '\n';
```

or

```
date aDate;

cout << '\n' << "Please, enter a date (day month year)" << '\n';
cin  >> aDate;
cout << aDate << '\n';
```

For the next step, we would implement shift operators as inserters and extractors for `date` objects. Here is an extractor for class `date`:

```
template<class charT, class Traits>
basic_istream<charT, Traits>&                                   //1
operator>> (basic_istream<charT,Traits>& is,                    //2
            date& dat)                                          //3
{
 is >> dat.tm_date.tm_mday;                                     //4
 is >> dat.tm_date.tm_mon;
 is >> dat.tm_date.tm_year;
 return is;                                                     //5
}
```

`//1` The returned value for extractors (and inserters) is a reference to the stream, so that several extractions can be done in one expression.

`//2` The first parameter usually is the stream from which the data shall be extracted.

`//3` The second parameter is a reference, or alternatively a pointer, to an object of the user-defined type.

`//4` In order to allow access to private data of the date class, the extractor must be declared as a friend function in class `date`.

`//5` The return value is the stream from which the data was extracted.

As the extractor accesses private data members of class date, it must be declared as a friend function to class date. The same holds for the inserter. Here's a more complete version of class `date`:

```
class date {
public:
   date(int d, int m, int y);
   date(tm t);
   date();
   // more constructors and useful member functions

private:
   tm tm_date;
```

```
template<class charT, Traits>
friend basic_istream<charT, Traits> &operator >>
  (basic_istream<charT, Traits >& is, date& dat);

template<class charT, Traits>
friend basic_ostream<charT, Traits> &operator <<
  (basic_ostream<charT, Traits >& os, const date& dat);
};
```

The inserter can be built analogously, as shown in the following code. The only difference is that you would hand over a *constant* reference to a date object, because the inserter is not supposed to modify the object it prints.

```
template<class charT, class Traits>
basic_ostream<charT, Traits>&
operator << (basic_ostream<charT, Traits >& os, const date& dat)
{
 os << dat.tm_date.tm_mon << '-';
 os << dat.tm_date.tm_mday << '-';
 os << dat.tm_date.tm_year ;
 return os;
}
```

2.7.3 Improved Extractors and Inserters

The format of dates depends on local and cultural conventions. Naturally, we want our extractor and inserter to parse and format the date according to such conventions. To add this functionality to them, we use the time facet contained in the respective stream's locale as follows:

```
template<class charT, class Traits>
basic_istream<charT, Traits>&
operator >> (basic_istream<charT, Traits >& is, date& dat)
{
 ios_base::iostate err = 0;

 use_facet<time_get<charT,Traits> >(is.getloc())              //1
   .get_date(is, istreambuf_iterator<charT,Traits>()          //2
            ,is, err, &dat.tm_date);                          //3

 return is;
}
```

`//1` Use the `time_get` facet of the input stream's locale to handle parsing of dates according to cultural conventions defined by the locale. The locale in question is obtained through the stream's `getloc()` function. Its `time_get` facet is accessed through a call to the global `use_facet<..>()` function. The type argument to the `use_facet` function template is the facet type. (See the chapter on internationalization for more details on locales and facets.).

`//2` The facet's member function `get_date()` is called. It takes a number of arguments, including:

A range of input iterators. For the sake of performance and efficiency, facets directly operate on a stream's buffer. They access the stream buffer through stream buffer iterators. (See the section on stream buffer iterators in the *Standard C++ Library User's Guide*.) Following the

philosophy of iterators in the Standard C++ Library, we must provide a range of iterators. The range extends from the iterator pointing to the first character to be accessed, to the character past the last character to be accessed (the *past-the-end-position*).

The beginning of the input sequence is provided as a reference to the stream. The `istreambuf_iterator` class has a constructor taking a reference to an input stream. Therefore, the reference to the stream is automatically converted into an `istreambuf_iterator` that points to the current position in the stream. As end of the input sequence, an end-of-stream iterator is provided. It is created by the default constructor of class `istreambuf_iterator`. With these two stream buffer iterators, the input is parsed from the current position in the input stream until a date or an invalid character is found, or the end of the input stream is reached.

`//3` The other parameters are:

Formatting flags. A reference to the `ios_base` part of the stream is provided here, so that the facet can use the stream's formatting information through the stream's members `flags()`, `precision()`, and `width()`.

An iostream state. It is used for reporting errors while parsing the date.

A pointer to a time object. It has to be a pointer to an object of type `tm`, which is the time structure defined by the C library. Our date class maintains such a time structure, so we hand over a pointer to the respective data member `tm_date`.

The inserter is built analogously:

```
template<class charT, class Traits>
basic_ostream<charT, Traits>& operator <<
 (basic_ostream<charT, Traits >& os, const date& dat)
{
   use_facet
   <time_put<charT,ostreambuf_iterator<charT,Traits> > >      //1
   (os.getloc())
   .put(os,os,os.fill(),&dat.tm_date,'x');                    //2
   return os;
}
```

`//1` Here we use the `time_put` facet of the stream's locale to handle formatting of dates.

`//2` The facet's `put()` function takes the following arguments:

An output iterator. We use the automatic conversion from a reference to an output stream to an `ostreambuf_iterator`. This way the output will be inserted into the output stream, starting at the current write position.

The formatting flags. Again we provide a reference to the `ios_base` part of the stream to be used by the facet for retrieving the stream's formatting information.

The fill character. We would use the stream's fill character here. Naturally, we could use any other fill character; however, the stream's settings are normally preferred.

A pointer to a time structure. This structure will be filled with the result of the parsing.

A format specifier. This can be a character, like `'x'` in our example here, or alternatively, a character sequence containing format specifiers, each consisting of a `%` followed by a character. An example of such a format specifier string is `"%A, %B %d, %Y"`. It has the same effect as the format specifiers for the `strftime()` function in the C library; it produces a date like: `Tuesday, June 11, 1996`. We don't use a format specifier string here, but simply the character `'x'`, which specifies that the locale's appropriate date representation shall be used.

Note how these versions of the inserter and extractor differ from previous simple versions: we no longer rely on existing inserters and extractors for built-in types, as we did when we used `operator<<(int)` to insert the `date` object's data members individually. Instead, we use a low-level service like the `time` facet's `get_date()` service. The consequence is that we give away all the functionality that high-level services like the inserters and extractors already provide, such as format control, error handling, etc.

The same happens if you decide to access the stream's buffer directly, perhaps for optimizing your program's runtime efficiency. The stream buffer's services, too, are low-level services that leave to you the tasks of format control, error handling, etc.

In the following sections, we will explain how you can improve and complete your inserter or extractor if it directly uses low-level components like locales or stream buffers.

2.7.4 More Improved Extractors and Inserters

Insertion and extraction still do not fit seamlessly into the iostream framework. The inserters and extractors for built-in types can be controlled through formatting flags that our operators thus far ignore. Our operators don't observe a field width while inserting, or skip whitespaces while extracting, and so on.

They don't care about error indication either. So what if the extracted date is February 31? So what if the insertion fails because the underlying buffer can't access the external device for some obscure reason? So what if a facet throws an exception? We should certainly set some state bits in the respective stream's state and throw or rethrow exceptions, if the exception mask says so.

However, the more general question here is: what are inserters and extractors supposed to do? Some recommendations follow.

- ***Regarding* format flags*, inserters and extractors should:***
  - Create a sentry object right at the beginning of every inserter and extractor. In its constructor and destructor, the sentry performs certain standard tasks, like skipping white characters, flushing tied streams, etc. See the *Class Reference* for a detailed explanation.
  - Reset the width after each usage.
- ***Regarding* state bits*, inserters and extractors should:***
  - Set `badbit` for all problems with the stream buffer.
  - Set `failbit` if the formatting or parsing itself fails.
  - Set `eofbit` when the end of the input sequence is reached.
- ***Regarding the* exception mask*, inserters and extractors should:***
  - Use the `setstate()` function for setting the stream's error state. It automatically throws the `ios_base::failure` exception according to the exceptions switch in the stream's exception mask.
  - Catch exceptions thrown during the parsing or formatting, set `failbit` or `badbit`, and rethrow the *original* exception.
- ***Regarding* locales*, inserters and extractors should:***
  - Use the stream's locale, not the stream buffer's locale. The stream buffer's locale is supposed to be used solely for code conversion. Hence, imbuing a stream with a new locale will only affect the stream's locale and never the stream buffer's locale.[30]
- ***Regarding the* stream buffer*:***
  - If you use a sentry object in your extractor or inserter, you should not call any functions from the formatting layer. This would cause a dead-lock in a multithreading situation, since the sentry object locks the stream through the stream's *mutex* (= mutual exclusive lock). A nested call to one of the stream's member functions would again create a sentry object, which would wait for the same mutually exclusive lock and, voilà, you have deadlock. Use the stream buffer's functions instead. They do not use the stream's mutex, and are more efficient anyway.

***Please note:* Do not call the stream's input or output functions after creating a sentry object in your inserter or extractor. Use the stream buffer's functions instead.**

---

[30] This, however, is still subject to change. At the time of this writing, a call to the stream's `imbue()` function modifies both locales.

2.7.4.1 Applying the Recommendations to the Example

Let us now go back and apply the recommendations to the extractor and inserter for class `date` in the example we have been constructing. Here is an improved version of the extractor:

```
template<class charT, class Traits>
basic_istream<charT, Traits>& operator >>
  (basic_istream<charT, Traits >& is, date& dat)
{
 ios_base::iostate err = 0;                                          //1

 try {                                                               //2

  typename basic_istream<charT, Traits>::sentry ipfx(is);            //3

  if(ipfx)                                                           //4
  {
   use_facet<time_get<charT,Traits> >(is.getloc())
        .get_date(is, istreambuf_iterator<charT,Traits>()
                  ,is, err, &dat.tm_date);                         //5
   if (!dat) err |= ios_base::failbit;                             //6
  }
 } // try
 catch(...)                                                          //7
 {
    bool flag = FALSE;
    try { is.setstate(ios_base::failbit); }                        //8
    catch( ios_base::failure ) { flag= TRUE; }                     //9
    if ( flag ) throw;                                            //10
 }

 if ( err ) is.setstate(err);                                       /11

 return is;
}
```

`//1` The variable `err` will keep track of errors as they occur. In this example, it is handed over to the `time_get` facet, which will set the respective state bits.

`//2` All operations inside an extractor or inserter should be inside a try-block, so that the respective error states could be set correctly before the exception is actually thrown.

`//3` Here we define the sentry object that does all the preliminary work, like skipping leading white spaces.

`//4` We check whether the preliminaries were done successfully. Class `sentry` has a conversion to `bool` that allows this kind of check.

`//5` This is the call to the time parsing facet of the stream's locale, as in the primitive version of the extractor.

`//6` Let's assume our date class allows us to check whether the date is semantically valid, e.g., it would detect wrong dates like February 30. Extracting an invalid date should be treated as a failure, so we set the `failbit`.

Note that in this case it is not advisable to set the `failbit` through the stream's `setstate()` function, because `setstate()` also raises exceptions if they are switched on in the stream's exception mask. We don't want to throw an exception at this point, so we add the `failbit` to the state variable `err`.

`//7` Here we catch all exceptions that might have been thrown so far. The intent is to set the stream's error state before the exception terminates the extractor, and to rethrow the original exception.

`//8` Now we eventually set the stream's error state through its `steatite()` function. This call might throw an `ios_base::failure` exception according to the stream's exception mask.

`//9` We catch this exception because we want the original exception thrown rather than the `ios_base::failure` in all cases.

`//10` We rethrow the original exception.

`//11` If there was no exception raised so far, we set the stream's error state through its `steatite()` function.

The inserter is implemented using the same pattern:

```
template<class charT, class Traits>
basic_ostream<charT, Traits>& operator <<
 (basic_ostream<charT, Traits >& os, const date& dat)
{
 ios_base::iostate err = 0;

 try {
  typename basic_ostream<charT, Traits>::sentry opfx(os);

  if(opfx)
  {
   char patt[3] = "%x";
   charT fmt[3];
   use_facet<ctype<charT> >(os.getloc())
      .widen(patt,patt+2,fmt);                                  //1
   if (
   use_facet<time_put<charT,ostreambuf_iterator<charT,Traits> > >
     (os.getloc())
     .put(os,os,os.fill(),&dat.tm_date,fmt,(fmt+2))             //2
     .failed()                                                  //3
   )
      err = ios_base::badbit;                                   //4
   os.width(0);                                                 //5
  }
 } //try
 catch(...)
 {
    bool flag = FALSE;
    try {
           os.setstate(ios_base::failbit);
        }
    catch( ios_base::failure ) { flag= TRUE; }
    if ( flag ) throw;
 }
```

```
  if ( err ) os.setstate(err);

  return os;
}
```

The inserter and the extractor have only a few minor differences:

`//1` We prefer to use the other `put()` function of the locale's `time_put` facet. It is more flexible and allows us to specify a sequence of format specifiers instead of just one. We declare a character array that contains the sequence of format specifiers and *widen* it to wide characters, if necessary.

`//2` Here we provide the format specifiers to the `time_put` facet's `put()` function.

`//3` The `put()` function returns an iterator pointing immediately after the last character produced. We check the success of the previous output by calling the iterators `failed()` function.

`//4` If the output failed then the stream is presumably broken, and we set `badbit`.

`//5` Here we reset the field width, because the facet's `put()` function uses the stream's format settings and adjusts the output according to the respective field width. The rule is that the field width shall be reset after each usage.

2.7.4.2 An Afterthought

Why is it seemingly so complicated to implement an inserter or extractor? Why doesn't the first simple approach suffice?

First, it is not really as complicated as it seems if you stick to the patterns: we give these patterns in the next section. Second, the simple extractors and inserters in our first approach do suffice in many cases, when the user-defined type consists mostly of data members of built-in types, and runtime efficiency is not a great concern.

However, whenever you care about the runtime efficiency of your input and output operations, it is advisable to access the stream buffer directly. In such cases, you will be using fast low-level services and hence will have to add format control, error handling, etc., because low-level services do not handle this for you. In our example, we aimed at optimal performance; the extractor and inserter for locale-dependent parsing and formatting of dates are very efficient because the facets directly access the stream buffer. In all these cases, you should follow the patterns we are about to give.

2.7.5 Patterns for Extractors and Inserters of User-Defined Types

Here is the pattern for an extractor:

```
template<class charT, class Traits>
basic_istream<charT, Traits>& operator >>
  (basic_istream<charT, Traits >& is, UserDefinedType& x)
{
 ios_base::iostate err = 0;

 try {

  typename basic_istream<charT, Traits>::sentry ipfx(is);

  if(ipfx)
  {
   // Do whatever has to be done!
   // Typically you will access the stream's locale or buffer.
   // Don't call stream member functions here in MT environments!

   // Add state bits to the err variable if necessary, e.g.
   // if (…) err |= ios_base::failbit;
  }
 } // try
 catch(...)                                                    //7
 {
    bool flag = FALSE;
    try { is.setstate(ios_base::failbit); }                    //8
    catch( ios_base::failure ) { flag= TRUE; }                 //9
    if ( flag ) throw;                                        //10
 }

 if ( err ) is.setstate(err);                                  /11

 return is;
}
```

Similarly, the pattern for the inserter looks like this:

```
template<class charT, class Traits>
basic_ostream<charT, Traits>& operator <<
 (basic_ostream<charT, Traits >& os, const UserDefinedType& x)
{
 ios_base::iostate err = 0;

 try {
  typename basic_ostream<charT, Traits>::sentry opfx(os);

  if(opfx)
  {
   // Do whatever has to be done!
   // Typically you will access the stream's locale or buffer.
   // Don't call stream member functions here in MT environments!

   // Add state bits to the err variable if necessary, e.g.
   // if (…) err |= ios_base::failbit;

   // Reset the field width after usage, i.e.
   // os.width(0);
  }
 } //try
```

```
  catch(...)
  {
     bool flag = FALSE;
     try { os.setstate(ios_base::failbit); }
     catch( ios_base::failure ) { flag= TRUE; }
     if ( flag ) throw;
  }

  if ( err ) os.setstate(err);

  return os;
}
```

2.8 Manipulators

We have seen examples of manipulators in Section 2.3.3.2. There we learned that:

- Manipulators are objects that can be inserted into or extracted from a stream, and
- Such insertions and extractions have specific desirable side effects.

As a recap, here is a typical example of two manipulators:

```
cout << setw(10) << 10.55 << endl;
```

The inserted objects `setw(10)` and `endl` are the manipulators. As a side effect, the manipulator `setw(10)` sets the stream's field width to `10`. Similarly, the manipulator `endl` inserts the end of line character and flushes the output.

As we have mentioned previously, extensibility is a major advantage of iostreams. We've seen in the previous section how you can implement inserters and extractors for user-defined types that behave like the built-in input and output operations. Additionally, you can add user-defined manipulators that fit seamlessly into the iostreams framework. In this section, we will see how to do this.

First of all, to be extracted or inserted, a manipulator must be an object of a type that we call `manipT`, for which overloaded versions of the shift operators exist. (Associated with the manipulator type `manipT`, there is usually a function that we will call $f_{manipT}()$ that we will explain in detail later.) Here's the pattern for the manipulator extractor:

```
template <class charT, class Traits>
basic istream<charT,Traits>&
operator>> (basic istream<charT,Traits>& istr
           ,const manipT& manip)
{   return f_manipT(istr, …);   }
```

With this extractor defined, you can extract a manipulator `Manip`, which is an object of type `manipT`, by simply saying:

```
cin >> Manip;
```

This results in a call to the `operator>>()` sketched out above. The manipulator inserter is analogous.

A manipulator's side effect is often created by calling an associated function $f_{manipT}()$ that takes a stream and returns the stream. There are several ways to associate the manipulator type `manipT` to the function $f_{manipT}()$ that we will explore in the subsequent sections. The iostream framework does not specify a way to implement manipulators, but there is a base class called `smanip` that you can use for deriving your own manipulators. We will explain this technique along with other useful approaches.

It will turn out that there is a major difference between manipulators with parameters like `width(10)` and manipulators without parameters like `endl`. Let's start with the simpler case of manipulators without parameters.

2.8.1 Manipulators without Parameters

Manipulators that do not have any parameters, like `endl`, are the simplest form of manipulator. The manipulator type `manipT` is a function pointer type, the manipulator `Manip` is the function pointer, and the associated function $f_{manipT}()$ is the function pointed to.

In iostreams, the following function pointer types serve as manipulators:

```
(1) ios_base&                    (*pf)(ios_base&)
(2) basic_ios<charT,Traits>&     (*pf)(basic_ios<charT,Traits>)
(3) basic_istream<charT,Traits>& (*pf)(basic_istream<charT,Traits>)
(4) basic_ostream<charT,Traits>& (*pf)(basic_ostream<charT,Traits>)
```

The signature of a manipulator function is not limited to the examples above. If you have created your own stream types, you will certainly want to use additional signatures as manipulators.

For the four manipulator types listed above, the stream classes already offer the required overloaded inserters and member functions. For input streams, extractors take the following form:

```
template<class charT, class traits>
basic_istream<charT, traits>&
basic_istream<charT, traits>::operator>>
(basic_istream<charT,traits>& (*pf)(input_stream_type&) )
{  return (*pf)(*this);..}
```

where ***input_stream_type*** is one of the function pointer types (1)-(3).

Similarly, for output streams we have:

```
template<class charT, class traits>
basic_ostream<charT, traits>&
basic_ostream<charT, traits>::operator<<
(basic_ostream<charT, traits>& (*pf)(output_stream_type& ))
{  return (*pf)(*this);  }
```

where ***output_stream_type*** is one of the function pointer types (1), (2), or (4).

2.8.1.1 Examples of Manipulators without Parameters

Let's look at the manipulator endl as an example of a manipulator without parameters. The manipulator endl, which can be applied solely to output streams, is a pointer to the following function of type (4):

```
template<class charT, class traits>
inline basic_ostream<charT, traits>&
endl(basic_ostream<charT, traits>& os)
{
  os.put( os.widen('\n') );
  os.flush();

  return os;
}
```

Hence an expression like:

```
cout << endl;
```

results in a call to the inserter:

```
ostream& ostream::operator<< (ostream& (*pf)(ostream&))
```

with endl as the actual argument for pf. In other words, cout << endl; is equal to cout.operator<<(endl);

Here is another manipulator, boolalpha, that can be applied to input *and* output streams. The manipulator boolalpha is a pointer to a function of type (1):

```
ios_base& boolalpha(ios_base& strm)
{
  strm.setf(ios_base::boolalpha);

  return strm;
}
```

Summary: **Every function that takes a reference to an ios_base, a basic_ios, a basic_ostream, or a basic_istream, and returns a reference to the same stream, can be used as a parameter-less manipulator.**

2.8.1.2 A Remark on the Manipulator endl

The manipulator endl is often used for inserting the end-of-line character into a stream. However, endl does additionally flush the output stream, as you can see from the implementation of endl shown above. Flushing a stream, a time-consuming operation that decreases performance, is unnecessary in most common situations. In the standard example:

```
cout << "Hello world" << endl;
```

flushing is not necessary because the standard output stream cout is tied to the standard input stream cin, so input and output to the standard streams are synchronized anyway. Since no flush is required, the intent is probably to insert the end-of-line character. If you consider typing '\n' more trouble

than typing endl, you can easily add a simple manipulator nl that inserts the end-of-line character, but refrains from flushing the stream.

2.8.2 Manipulators with Parameters

Manipulators with parameters are more complex than those without because there are additional issues to consider. Before we explore these issues in detail and examine various techniques for implementing manipulators with parameters, let's take a look at one particular technique, the one that is used to implement standard manipulators such as setprecision(), setw(), etc.

2.8.2.1 The Standard Manipulators

Rogue Wave's implementation of the standard iostreams uses a certain technique for implementing most standard manipulators with parameters: the manipulator type manipT is a function pointer type; the manipulator object is the function pointed to; and the associated function f_{manipT} is a global function.

The C++ standard defines the manipulator type as smanip. The type itself is implementation-defined; all you know is that it is returned by some of the standard manipulators. In Rogue Wave's implementation, smanip is a class template:

```
template<class T>
class smanip {
  public:
    smanip(ios_base& (*pf)(ios_base&, T), T manarg);
};
```

A standard manipulator like setprecision() can be implemented as a global function returning an object of type smanip<T>:

```
inline smanip<int> setprecision(int n)
{ return smanip<int>(sprec, n); }
```

The associated function f_{manipT} is the global function sprec:

```
inline ios_base& sprec(ios_base& str, int n)
{
  str.precision(n);
  return str;
}
```

2.8.2.2 The Principle of Manipulators with Parameters

The previous section gave an example of a technique for implementing a manipulator with one parameter: the technique used to implement the standard manipulators in iostreams. However, this is not the only way to implement a manipulator with parameters. In this section, we examine other techniques. Although all explanations are in terms of manipulators with one

parameter, it is easy to extend the techniques to manipulators with several parameters.

Let's start right at the beginning: what is a manipulator with a parameter?

A manipulator with a parameter is an object that can be inserted into or extracted from a stream in an expression like:

```
cout << Manip(x);
cin  >> Manip(x);
```

`Manip(x)` must be an object of type `manipT`, for which the shift operators are overloaded. Here's an example of the corresponding inserter:

```
template <class charT, class Traits>
basic ostream<charT,Traits>&
operator<< (basic ostream<charT,Traits>& ostr
           ,const manipT& manip)
{  // call the associated function f_manipT, e.g
   (*manip.f_manipT )(ostr,manip.arg_f);
   return os;
}
```

With this inserter defined, the expression `cout << Manip(x);` is equal to a call to the shift operator sketched above; i.e., `operator<<(cout, Manip(x));`

Assuming that a side effect is created by an associated function f_{manipT}, the manipulator must call the associated function with its respective argument(s). Hence it must store the associated function together with its argument(s) for a call from inside the shift operator.

The associated function f_{manipT} can be a static or a global function, or a member function of type `manipT`, for example.

In the inserter above, we've assumed that the associated function f_{manipT} is a static or a global function, and that it takes exactly one argument. Generally, the manipulator type `manipT` might look like this:

```
template <class FctPtr, class Arg1, class Arg2, …>
class manipT
{
 public:
   manipT(FctPtr, Arg1, Arg2, …);
 private:
   FctPtr fp_;
   Arg1   arg1_;
   Arg2   arg2_;
};
```

Note that this is only a suggested manipulator, however. In principle, you can define the manipulator type in any way that makes the associated side effect function and its arguments available for a call from inside the respective shift operators for the manipulator type. We will see other examples of such manipulator types later in this section; for instance, a manipulator type called `smanip` defined by the C++ standard. It is an implementation-defined function type returned by the standard manipulators. See the *Class Reference* for details.

Returning now to the example above, the manipulator object provided as an argument to the overloaded shift operator is obtained by `Manip(x)`, which has three possible solutions:

(1) `Manip(x)` is a function call. In this case, `Manip` would be the name of a function that takes an argument of type `x` and returns a manipulator object of type `manipT`; i.e., `Manip` is a function with the following signature:

```
manipT Manip (X x);
```

(2) `Manip(x)` is a constructor call. In this case, `Manip` would be the name of a class with a constructor that takes an argument of type `X` and constructs a manipulator object of type `Manip`; i.e., `Manip` and `manipT` would be identical:

```
class Manip {
 public:
   Manip(X x);
};
```

(3) `Manip(x)` is a call to a function object. In this case, `Manip` would be an object of a class `M`, which defines a function call operator that takes an argument of type `x` and returns a manipulator object of type `manipT`:

```
class M {
 public:
   manipT operator()(X x);
} Manip;
```

Solutions (1) and (2) are semantically different from solution (3). In solution (1), `Manip` is a function and therefore need not be created by the user. In solution (2), `Manip` is a class name and an unnamed temporary object serves as manipulator object. In solution (3), however, the manipulator object `Manip` must be explicitly created by the user. Hence the user has to write:

```
manipT Manip;
cout << Manip(x);
```

which is somewhat inconvenient because it forces the user to know an additional name, the manipulator type `manipT`, and to create the manipulator object `Manip`.[31] For these reasons, solution (3) is useful if the manipulator has *state*; i.e., if it stores additional data like a manipulator, let call it `lineno`, which provides the next line number each time it is inserted.

The problem with solution (2) is that several current compilers cannot handle unnamed objects.

---

[31] An alternative could be to provide `Manip` as a static or a global object at the user's convenience. Unfortunately, this approach would introduce the well-known order-of-initialization problems for global and static objects.

For any of the three solutions just discussed, there is also a choice of associated functions. The associated function f_{manipT} can be either:

a) A static or a global function;

b) A static member function;

c) A virtual member function.

Among these choices, (b), i.e. use of a static member function, is the preferable in an object-oriented program because it permits encapsulation of the manipulator together with its associated function. This is particularly recommended if the manipulator has *state*, as in solution (3), where the manipulator is a function object, and the associated function has to access the manipulator's state.

Using (c), i.e. a virtual member function, introduces the overhead of a virtual function call each time the manipulator is inserted or extracted. It is useful if the manipulator has state, and the state needs to be modified by the associated manipulator function. A static member function would only be able to access the manipulator's static data; a non-static member function, however, can access the object-specific data.

2.8.2.3 Examples of Manipulators with Parameters

In this section, let's look at some examples of manipulators with parameters. The examples here are arbitrary combinations of solutions (1) to (3) for the manipulator type, with (a) to (c) for the associated function. We also use the standard manipulator `setprecision()`to demonstrate the various techniques.

Example 1: Function Pointer and Global Function. This example combines (1) and (c), and so:

- `manipT` is a function pointer type;
- The manipulator object is the function pointed to; and
- The associated function f_{manipT} is a global function.

Rogue Wave's implementation of the standard iostreams uses this technique for implementing most standard manipulators with parameters. See Section 2.8.2.1 for reference.

Example 2: Unnamed Object and Static Member Function. This example combines (2) and (b), and thus:

- The manipulator object `Manip` is an unnamed object;
- The manipulator type `manipT` is a class; and
- The associated function f_{manipT} is a static member function.

The manipulator type `manipT` can be derived from the manipulator type `smanip` defined by iostreams. Here is an alternative implementation of a manipulator like `setprecision()`:

```
class setprecision : public smanip<int> {
  public:
    setprecision(int n) : smanip<int>(sprec_, n) { }
  private:
    static ios_base& sprec_(ios_base& str, int n)
    { str.precision(n);
      return str;
    }
};
```

Example 3: Unnamed Object and Virtual Member Function. This example (2) and (c), and therefore:

- The manipulator object `Manip` is an unnamed object;
- The manipulator type `manipT` is a class; and
- The associated function f_{manipT} is a virtual member function of that class.

The idea here is that the associated function f_{manipT} is a non-static member function of the manipulator type `manipT`. In such a model, the manipulator does not store a pointer to the associated function f_{manipT}, but defines the associated function as a pure virtual member function. Consequently, the manipulator type `manipT` will be an abstract class, and concrete manipulator types will be derived from this abstract manipulator type. They will have to implement the virtual member function that represents the associated function.

Clearly, we need a new manipulator type because the standard manipulator type `smanip` is implementation-defined. In Rogue Wave's *Standard C++ Library*, it has no virtual member functions, but stores a pointer to the associated function. Here is the abstract manipulator type we need:

```
template <class Arg, class Ostream>
class virtsmanip
{
 public:
  typedef Arg argument_type;
  typedef Ostream ostream_type;
  virtsmanip (Arg a) : arg_(a) { }

 protected:
  virtual Ostream& fct_(Ostream&,Arg) const = 0;
  Arg arg_;

  friend Ostream&
  operator<< (Ostream& ostr
              ,const virtsmanip<Arg,Ostream>& manip);
};
```

This type `virtsmanip` differs from the standard type `smanip` in several ways:

- It defines the above-mentioned pure virtual member function `fct_()`.
- The argument `arg_` and the virtual function `fct_()` are protected members, and consequently the respective shift operator for the manipulator type has to be a friend function.
- It is a base class for output manipulators only.

The standard manipulator `smanip` expects a pointer to a function that takes an `ios_base` reference. In this way, a manipulator is always applicable to input *and* output streams, regardless of whether or not this is intended. With our new manipulator type `virtsmanip,` we can define manipulators that cannot inadvertently be applied to input streams.

Since we have a new manipulator type, we also need a new overloaded version of the manipulator inserter:

```
template <class Arg, class Ostream>
Ostream&
operator<< (Ostream& ostr, const virtsmanip<Arg,Ostream>& manip)
{
   manip.fct_(ostr,manip.arg_);
   return ostr;
}
```

After these preparations, we can now provide yet another alternative implementation of a manipulator like `setprecision()`. This time `setprecision()` is a manipulator for output streams only:

```
class setprecision : public virtsmanip<int,basic_ostream<char> >
{
public:
 setprecision(argument_type n)
 : virtsmanip<argument_type,ostream_type>(n) { }

protected:
 ostream_type& fct_(ostream_type& str, argument_type n) const
 {
   str.precision(n);
   return str;
 }
};
```

Example 4: Function Object and Static Member Function. The next example combines (3) and (b), so here:

- The manipulator object `Manip` is an object of a type `M` that defines the function call operator;
- The manipulator type `manipT` is a class type that is returned by the overloaded function call operator of class `M`; and
- The associated function f_{manipT} is a static member function of class `M`.

This solution, using a function object as a manipulator, is semantically different from the previous solution in that the manipulator object has *state*, i.e., it can store data between subsequent uses.

Let us demonstrate this technique in terms of another example: an output manipulator that inserts a certain string that is maintained by the manipulator object. Such a manipulator could be used, for instance, to insert a prefix to each line:

```
Tag<char> change_mark("v1.2 >> ");

while ( new_text )
   ostr << change_mark << next_line;

change_mark("");
while ( old_text)
   ostr << change_mark << next_line;
```

We would like to derive the `Tag` manipulator here from the standard manipulator `smanip`. Unfortunately, `smanip` is restricted to associated

functions that take an ios_base reference as a parameter. In our example, we want to insert the stored text to the stream, so we need the stream's inserter. However, ios_base does not have inserters or extractors. Consequently we need a new manipulator base type, similar to smanip, that allows associated functions that take a reference to an output stream:

```
template <class Ostream, class Arg>
class osmanip {
  public:
    typedef Ostream ostream_type;
    typedef Arg argument_type;

    osmanip(Ostream& (*pf)(Ostream&, Arg), Arg arg)
    : pf_(pf) , arg_(arg) { ; }

  protected:
    Ostream&     (*pf_)(Ostream&, Arg);
    Arg          arg_;

  friend Ostream&
    operator<<
    (Ostream& ostr, const osmanip<Ostream,Arg>& manip);
};
```

Then we need to define the inserter for the new manipulator type osmanip:

```
template <class Ostream, class Arg>
Ostream&
operator<< (Ostream& ostr,const osmanip<Ostream,Arg>& manip)
{
   (*manip.pf_)(ostr,manip.arg_);
   return ostr;
}
```

Now we define the function object type M, here called Tag:

```
template <class  charT>
class Tag
: public osmanip<basic_ostream<charT>, basic_string<charT> >
{
public:
 Tag(argument_type a = "")
  : osmanip<basic_ostream<charT>, basic_string<charT> >
    (fct_, a) { }

 osmanip<ostream_type,argument_type>&
  operator() (argument_type a)
  {
    arg_ = a;
    return *this;
  }

private:
 static ostream_type& fct_ (ostream_type& str, argument_type a)
 {
   return str << a;
 }
};
```

Note that the semantics of this type of manipulator differ from the previous ones, and from the standard manipulator setprecision. The manipulator

object has to be explicitly created before it can be used, as shown in the example below:

```
Tag<char> change_mark("v1.2 >> ");

while ( new_text )
   ostr << change_mark << next_line;

change_mark("");
while ( old_text)
   ostr << change_mark << next_line;
```

This kind of manipulator is more flexible. In the example above, you can see that the default text is set to `"v1.2 >> "` when the manipulator is created. Thereafter you can use the manipulator as a parameterless manipulator and it will remember this text. You can also use it as a manipulator taking an argument, and provide it with a different argument each time you insert it.

Example 5: Function Object and Virtual Member Function. In the previous example, a static member function is used as the associated function. This has the slight disadvantage that the associated function cannot modify the manipulator's state. Should modification be necessary, you might consider using a virtual member function instead.

Our final example here is a manipulator that stores additional data, the previously mentioned `lineno` manipulator. It adds the next line number each time it is inserted:

```
LineNo lineno;
while (!cout)
{
   cout << lineno << …;
}
```

The manipulator is implemented following the (3) and (b) pattern, i.e.:

- The manipulator object `Manip` is an object of a type `M` that defines the function call operator;
- The manipulator type `manipT` is a class type that is returned by the overloaded function call operator of class `M`; and
- The associated function f_{manipT} is a virtual member function of class `M`.

The manipulator object contains a line number that is initialized when the manipulator object is constructed. Each time the `lineno` manipulator is inserted, the line number is incremented.

For the manipulator base type, we use a slightly modified version of the manipulator type `osmanip` from Example 3. The changes are necessary because the associated function in this case may not be a constant member function:

```
emplate <class Arg, class Ostream>
class virtsmanip
{
 public:
  typedef Arg argument_type;
```

```
  typedef Ostream ostream_type;
  virtsmanip (Arg a) : arg_(a) { }

 protected:
  virtual Ostream& fct_(Ostream&,Arg) = 0;
  Arg arg_;

  friend Ostream&
  operator<< (Ostream& ostr
              ,virtsmanip<Arg,Ostream>& manip);
};

template <class Arg,class Ostream>
Ostream&
operator<< (Ostream& ostr
            ,virtsmanip<Arg,Ostream>& manip)
{
   manip.fct_(ostr,manip.arg_);
   return ostr;
}
```

The line number manipulator could be implemented like this:

```
template <class Ostream>
class LineNo
 : public virtsmanip<int,Ostream >
{
public:
 LineNo(argument_type n=0)
  : virtsmanip<argument_type, ostream_type> (n)
  { }

 virtsmanip<argument_type,ostream_type>&
  operator() (argument_type arg)
  {
   arg_ = arg;
   return *this;
  }
protected:
 argument_type lno_;
 ostream_type& fct_ (ostream_type& str, argument_type n)
 {
   lno_ = (n>0) ? n : lno_;
   str << ++lno_;
   arg_ = -1;
   return str;
 }
};
```

Using a virtual member function as the associated manipulator function introduces the overhead of a virtual function call each time the manipulator is inserted. If it is necessary that a manipulator update its state after each insertion, a static member function will not suffice. A global function that is a friend of the manipulator type might do the trick. However, in an object-oriented program, you are usually advised against global functions that modify private or protected data members of a class they are friends with.

2.9 Streams and Stream Buffers

So far we have used streams as the source or target of input and output operations. But there is another aspect of streams: they are also objects in your C++ program that you might want to copy and pass around like other objects. However, streams cannot simply be copied and assigned. The following section shows what you have to do instead.

Stream buffers play a special role. In numerous cases you will not want to create copies of a stream buffer object, but simply share a stream buffer among streams. Section 2.9.2 explores this option.

If you really want to work with copies of stream buffers, see section 2.9.3 for hints and caveats.

2.9.1 Copying and Assigning Stream Objects

Stream objects cannot simply be copied and assigned. Let us consider a practical example to see what it means. A program writes data to a file if a file name is specified on program call, or to the standard output stream `cout` if no file name is specified. You want to write to one output stream in your program; this stream will either be a file stream or the standard output stream. The most obvious way to do this is to declare an output file stream object and assign it to `cout`, or to use `cout` directly. However, you can't do it this way:

```
int main(int argc, char argv[])
{
  ofstream fil;
  if (argc > 1)
  {  fil.open(argv[1]);
     cout = fil;                          // never do this !!!
  }
  // output to cout, e.g.
  cout << "Hello world!" << endl;
}
```

This solution is bad for at least two reasons. First, assignments to any of the predefined streams should be avoided. The predefined stream `cin`, `cout`, `cerr`, and `clog` have special properties and are treated differently from other streams. If you reassign them, as done with `cout` in the example above, you lose their special properties. Second, assignment and copying of streams is hazardous. The assignment of the output stream `fil` will compile and might even work; however, your program is likely to crash afterwards.[32]

[32] Traditional iostreams had classes called ostream_withassign that explicitly allowed copying and assignment of stream objects.

Please note: **Stream objects must never be copied or assigned to each other.**

Let us see why. The copy constructor and the assignment operator for streams are not defined[33]; therefore, the compiler-generated default copy constructor and assignment operator are used. As usual, they only copy and assign a stream object's data members, which is insufficient because a stream object holds pointers as well. Figure 31 is a sketch of the data held by a stream object:

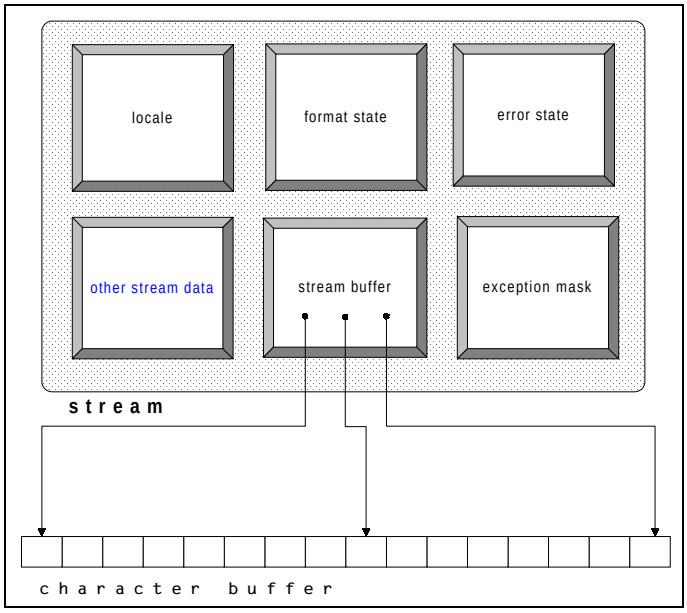


Figure 31. Data held by a stream object. Please note that some data members are omitted for the sake of simplicity.

The stream buffer contains several pointers to the actual character buffer it maintains. The default copy constructor and assignment operator will not correctly handle these pointers.

2.9.1.1 Copying a Stream's Data Members

---

[33] Traditional iostreams had the copy constructor and assignment operator defined as private member functions; hence, they were not usable.

It is an open issue whether the copy constructor and assignment operator of streams should be defined as private member functions.

To achieve the equivalent effect of a copy, you might consider copying each data member individually. This can be done as in the following example:

```
int main(int argc, char argv[])
{
  ofstream out;
  if (argc > 1)
     out.open(argv[1]);
  else
  {  out.copyfmt(cout);                                    //1
     out.clear(cout.rdstate());                            //2
     out.rdbuf(cout.rdbuf());                              //3
  }
  // output to out, e.g.
  out << "Hello world!" << endl;
}
```

`//1` The `copyfmt()` function copies all data from the standard output stream `cout` to the output file stream `out`, except the error state and the stream buffer. (There is a function `exceptions()` that allows you to copy the exception mask separately; i.e., `cout.exceptions(fil.exceptions());`. However, you need not do this explicitly, since `copyfmt()` already copies the exception mask.)

`//2` Here the error state is copied.

`//3` Here the stream buffer pointer is copied.

Please note the little snag here. After the call to `rdbuf()`, the buffer is shared between the two streams, as shown in Figure 32:

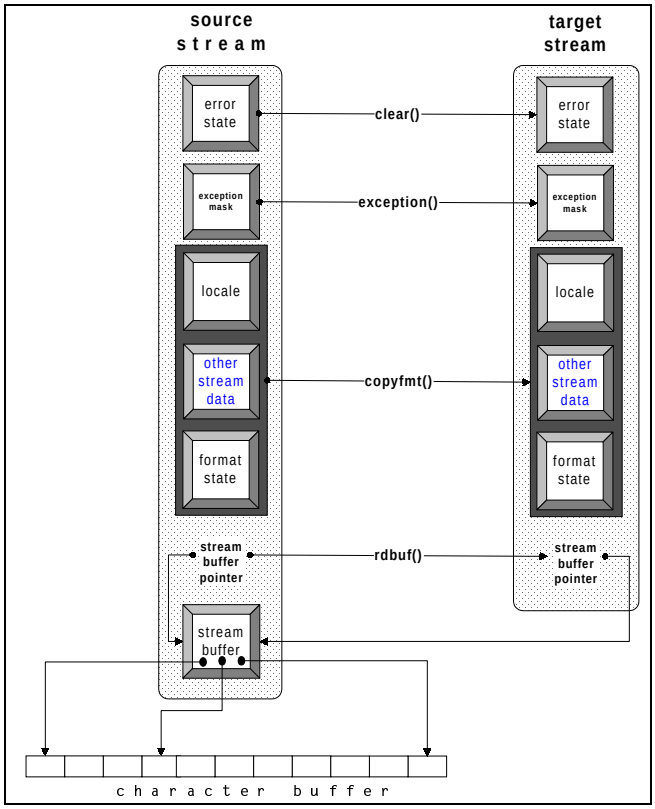


Figure 32. Copying a stream's internal data results in a shared buffer

2.9.1.2 Sharing Stream Buffers Inadvertently

Whether or not you intend to share a stream buffer among streams depends on your application. In any case, it is important that you realize the stream buffer is shared after a call to `rdbuf()`; in other words, you must monitor the lifetime of the stream buffer object and make sure it exceeds the lifetime of the stream. In our little example above, we use the standard output stream's buffer. Since the standard streams are static objects, their stream buffers have longer lifetimes that most other objects, so we are safe. However, whenever you share a stream buffer among other stream objects, you must carefully consider the stream buffer's lifetime.

The example above has another disadvantage we haven't considered yet, as shown in the following code:

```
int main(int argc, char argv[])
{
  ofstream out;
  if (argc > 1)
     out.open(argv[1]);
  else
  {  out.copyfmt(cout);                                    //1
     out.clear(cout.rdstate());                            //2
```

```
    out.rdbuf(cout.rdbuf());                                    //3
  }
  // output to out, e.g.
  out << "Hello world!" << endl;
}
```

As we copy the standard output stream's entire internal data, we also copy its special behavior. For instance, the standard output stream is synchronized with the standard input stream. (See Section 2.10.4 for further details.) If our output file stream `out` is a copy of `cout`, it is forced to synchronize its output operations with all input operations from `cin`. This might not be desired, especially since synchronization is a time-consuming activity. Here is a more efficient approach using only the stream buffer of the standard output stream:

```
int main(int argc, char argv[])
{
  filebuf* fb = new filebuf;                                    //1
  ostream out((argc>1)?                                         //2
    fb->open(argv[1],ios_base::out|ios_base::trunc):            //3
    cout.rdbuf());                                              //4
  if (out.rdbuf() != fb)
    delete fb;
  out << "Hello world!" << endl;
}
```

`//1` Instead of creating a file stream object, which already contains a file buffer object, we construct a separate file buffer object on the heap that we can hand over to an output stream object if needed. This way we can delete the file buffer object if not needed. In the original example, we constructed a file stream object with no chance of eliminating the file buffer object if not used.

`//2` An output stream is constructed. The stream has either the standard output stream's buffer, or a file buffer connected to a file.

`//3` If the program is provided with a file name, the file is opened and connected to the file buffer object. (Note that you must ensure that the lifetime of this stream buffer object exceeds the lifetime of the output stream that uses it.) The `open()` function returns a pointer to the file buffer object. This pointer is used to construct the output stream object.

`//4` If no file name is provided, the standard output stream's buffer is used.

As in the original example, `out` inserts through the standard output stream's buffer, but lacks the special properties of a standard stream.

Here is an alternative solution that uses file descriptors, a non-standard feature of Rogue Wave's implementation of the standard iostreams[34]:

---

[34] This feature was available in the traditional iostreams, but is not offered by the standard iostreams. Rogue Wave's implementation of the standard iostreams retains the old feature for backward compatibility with the traditional

```
int main(int argc, char argv[])
{
  ofstream out;
  if (argc > 1)     out.open(argv[1]);                    //1
  else              out.rdbuf()->open(1);                 //2
  out << "Hello world!" << endl;
}
```

`//1` If the program is provided with a file name, the file is opened and connected to the file buffer object.

`//2` Otherwise, the output stream's file buffer is connected to the standard input stream `stdout` whose file descriptor is `1`.

The effect is the same as in the previous solution, because the standard output stream `cout` is connected to the C standard input file `stdout`. This is the simplest of all solutions, because it doesn't involve reassigning or sharing stream buffers. The output file stream's buffer is simply connected to the right file. However, this is a non-standard solution, and may decrease portability.

2.9.1.3 Using Pointer or References to Streams

If you do not want to deal with stream buffers at all, you can also use pointers or references to streams instead. Here is an example:

```
int main(int argc, char argv[])
{
  ostream* fp;                                           //1
  if (argc > 1)
      fp = new ofstream(argv[1]);                        //2
  else
     fp = &cout                                          //3

  // output to *fp, e.g.
  *fp << "Hello world!" << endl;                         //4
  if(fp!=&cout) delete fp;
}
```

`//1` A pointer to an `ostream` is used. (Note that it cannot be a pointer to an `ofstream`, because the standard output stream `cout` is not a file stream, but a plain stream of type `ostream`.)

`//2` A file stream for the named output file is created on the heap and assigned to the pointer, in case a file name is provided.

`//3` Otherwise, a pointer to `cout` is used.

`//4` Output is written through the pointer to either `cout` or the named output file.

---

iostreams, but it is a non-standard feature. Using it might make your application non-portable to other standard iostream libraries.

An alternative approach could use a reference instead of a pointer:

```
int main(int argc, char argv[])
{
  ostream& fr = (argc > 1) ? *(new ofstream(argv[1])) : cout;
  // output to *fr, e.g.
  fr << "Hello world!" << endl;
  if (&fr!=&cout) delete(&fr);
}
```

Working with pointers and references has a drawback: you have to create an output file stream object on the heap and, in principle, you have to worry about deleting the object again, which might lead you into other dire straits.

In summary, creating a copy of a stream is not trivial and should only be done if you really need a copy of a stream object. In many cases, it is more appropriate to use references or pointers to stream objects instead, or to share a stream buffer between two streams.

Keep in mind: **Never create a copy of a stream object when a reference or a pointer to the stream object would suffice, or when a shared stream buffer would solve the problem.**

2.9.2 Sharing a Stream Buffer Among Streams

Despite the previous caveats, there are situations where sharing a stream buffer among streams is useful and intended. Let us focus on these in this section.

2.9.2.1 Several Format Settings for the Same Stream

Imagine you need different formatting for different kinds of output to the same stream. Instead of switching the format settings between the different kinds of output, you can arrange for two separate streams to share a stream buffer. The streams would have different format settings, but write output to the same stream buffer. Here is an example:

```
ofstream file1("/tmp/x");
ostream file2(file1.rdbuf());                                    \\1

file1.setf(ios_base::fixed, ios_base::floatfield);               \\2
file1.precision(5);
file2.setf(ios_base::scientific, ios_base::floatfield);
file2.precision(3);

file1 << setw(10) << 47.11 << '\n';                              \\3
file2 << setw(10) << 47.11 << '\n';                              \\4
```

`//1` The stream buffer of `file1` is replaced by the stream buffer of `file2`. Afterwards, both streams share the buffer.

`//2` Create different format settings for both files.

//3 The output here will be: 47.11000

//4 The output here will be: 4.711e+01

Note that file2 in the example above has to be an output stream rather than an output file stream. This is because file streams do not allow you to switch the file stream buffer.

2.9.2.2 Several Locales for the Same Stream

Similarly, you can use separate streams that share a stream buffer in order to avoid locale switches. This is useful when you have to insert multilingual text into the same stream. Here is an example:

```
ostringstream file1;
ostream file2(file1.rdbuf());

file1.imbue(locale("De_DE"));
file2.imbue(locale("En_US"));

file1 << 47.11 << '\t';
file2 << 47.11 << '\n';

cout << file1.str() << endl;                                    \\1
```

//1 The output will be: 47,11 47.11

Again, there is a little snag. In Figure 33, note that a stream buffer has a locale object of its own, in addition to the stream's locale object.

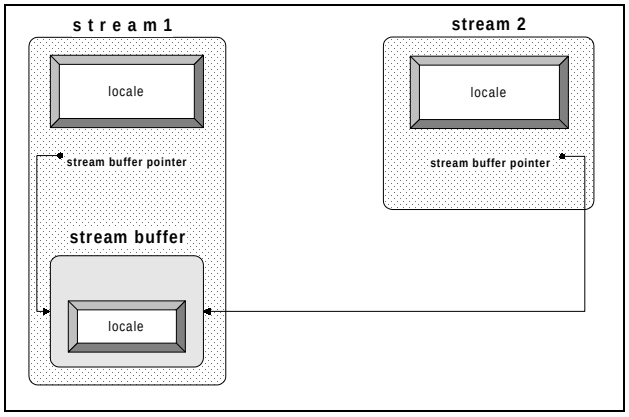


Figure 33. Locale objects and shared stream buffers

Section 2.2.2.1 explained the role of those two locale objects. To recap, the stream delegates the handling of numeric entities to its locale's numeric facets. The stream buffer uses its locale's code conversion facet for character-wise transformation between the buffer content and characters transported to and from the external device.

Usually the stream's locale and the stream buffer's locale are identical. However, when you share a stream buffer between two streams with different locales, you must decide which locale the stream buffer will use.[35]

You can set the stream buffer's locale by calling the `pubimbue()` function as follows:

```
..file1.imbue(locale("De_DE"));
  file2.imbue(locale("En_US"));
  file1.rdbuf()->pubimbue(locale("De_DE"));
```

2.9.2.3 Input and Output to the Same Stream

You can also use a shared stream buffer in order to have read *and* write access to a stream:

```
filebuf fbuf;                                                    //1
  fbuf.open("/tmp/inout",ios_base::in|ios_base::out);            //2
  istream in(&fbuf);                                             //3
  ostream out(&fbuf);                                            //4

  cout << in.rdbuf();                                            //5
  out << "..." << '\n' ;                                         //6
```

`//1` Create a file buffer, and

`//2` Connect it to a file. Note that you have to open the file in input and output mode if you want to read *and* write to it.

`//3` Create an input stream that works with the file buffer `fbuf`.

`//4` Create an output stream that also uses the file buffer `fbuf`.

`//5` Read the entire content of the file and insert it into the standard output stream. Afterwards the file position is at the end of the file.

The most efficient way to read a file's entire content is through the `rdbuf()` function, which returns a pointer to the underlying stream buffer object. There is an inserter available that takes a stream buffer pointer, so you can insert the buffer's content into another stream.

`//6` Write output to the file. As the current file position is the end of the file, all output will be inserted at the end.

---

[35] Whether and how the user can influence the setting of the stream buffer's locale is still open. At the time of this writing, a call to the stream's `imbue()` function changes the stream buffer's locale object as well. In the example above, the shared stream buffer will have the locale object of `file2`. This is not a problem here because the stream buffer only uses the locale's conversion facet, and both locales will probably have the same void conversion facet. However, this would cause a problem in the case where the streams' locales have different code conversion facets.

Naturally, it is easier and less error-prone to use bidirectional streams when you have to read and write to a file. The bidirectional equivalent to the example above would be:

```
fstream of("/tmp/inout");
  cout << of.rdbuf();
  of << "..." << '\n' ;
```

Notice that there is a difference between the solutions that you can see by comparing Figure 34 and Figure 35. An input and an output stream that share a stream buffer, as shown in Figure 34, can still have separate format settings, different locales, different exception masks, and so on.

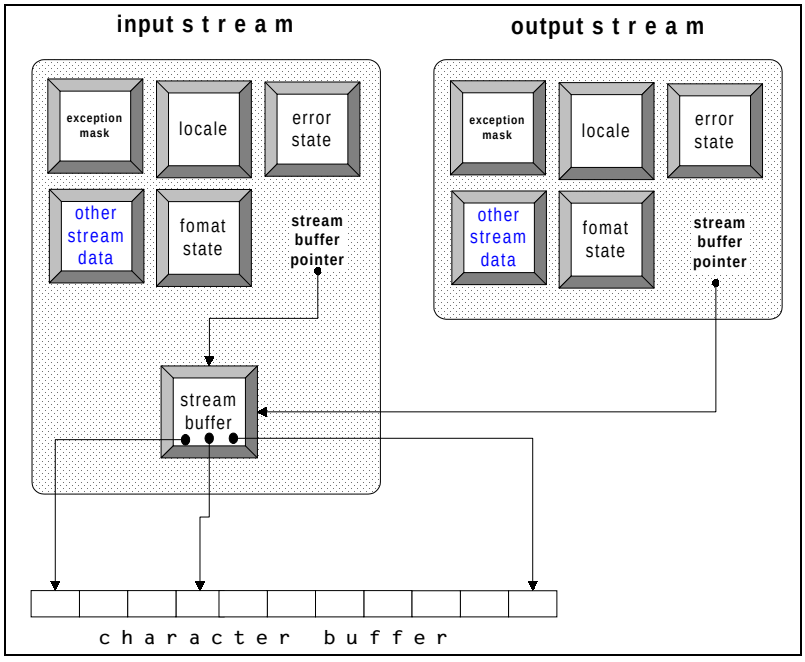


Figure 34. An input and an output stream sharing a stream buffer

In contrast, the bidirectional stream shown in Figure 35 can have only one format setting, one locale, and so on:

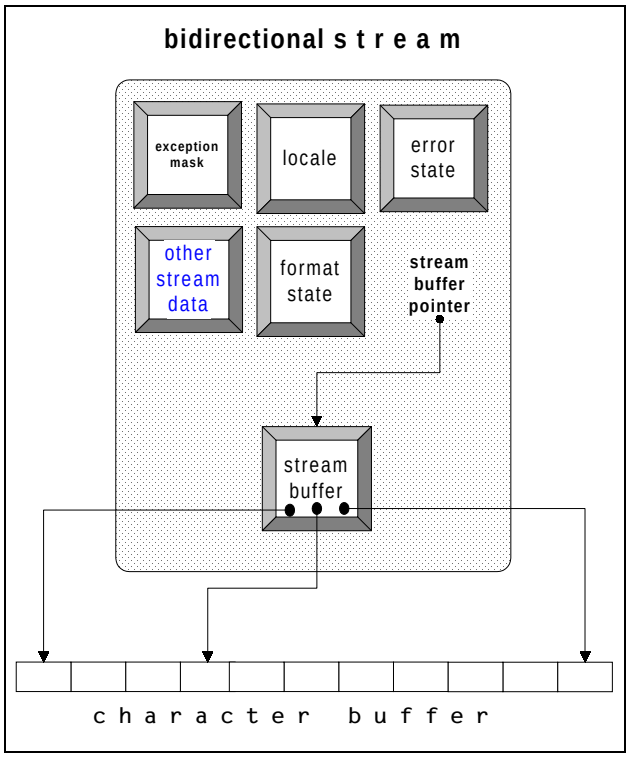


Figure 35. A bidirectional stream

It seems clear that you cannot have different settings for input and output operations when you use a bidirectional stream. Still, it is advisable to use bidirectional file or string streams if you need to read and write to a file or string, instead of creating an input and an output stream that share a stream buffer. The bidirectional stream is easier to declare, and you do not have to worry about the stream buffer object's lifetime.

Please note: **It's better to use one bidirectional file or string stream for reading and writing to a file or string, rather than two streams that share a stream buffer.**

2.9.3 Copies of the Stream Buffer

The previous section showed how you can read the content of a file in its entirety by using the `rdbuf()` function. Let us now explore a variation of that example. Imagine another file containing some sort of header information that needs to be analyzed before we start appending the file. Instead of writing the current file content to the standard output stream, we want to process the content before we start adding to it. The easiest way to put the

entire file content into a memory location for further processing is by creating a string stream and inserting the file's stream buffer into the string stream:

```
fstream fil("/tmp/inout");
  stringstream header_stream;                                    //1
  header_stream << fil.rdbuf();                                  //2

  // process the header, e.g.
  string word;
  header_stream >> word;                                         //3
```

`//1` The easiest way to put the entire file content into a memory location for further processing is by creating a string stream, and

`//2` Inserting the file's stream buffer into the string stream.

`//3` We now have the usual facilities of an input stream for reading and analyzing the header information; i.e., `operator>>()`, `read()`, `get()`, and so on.

In cases where this procedure is insufficient, you should create a string that contains the header information and process the header by means of the string operations `find()`, `compare()`, etc.

```
fstream fil("/tmp/inout");
  header_stream << fil.rdbuf();
  string header_string = header_stream.str();

  // process the header, e.g.
  string::size_type pos = header_string.rfind('.');
```

If the header contains binary data instead of text, even a string will probably not suffice. Here you would want to see the header as a plain byte sequence, i.e., an ordinary `char*` buffer. But note that a code conversion might already have been performed, depending on the locale attached to the file stream. In cases where you want to process binary data, you have to make sure that the attached locale has a non-converting code conversion facet:

```
fstream fil("/tmp/inout");
  header_stream << fil.rdbuf();
  string header_string = header_stream.str();
  const char* header_char_ptr = header_string.data();

  // process the header, e.g.
  int idx;
  memcpy((char*) &idx,header_char_ptr,sizeof(int));
```

A note on efficiency: If the header information is extensive, you will have to consider the number of copy operations performed in the previous example. ***Figure 36*** shows how these copies are made:

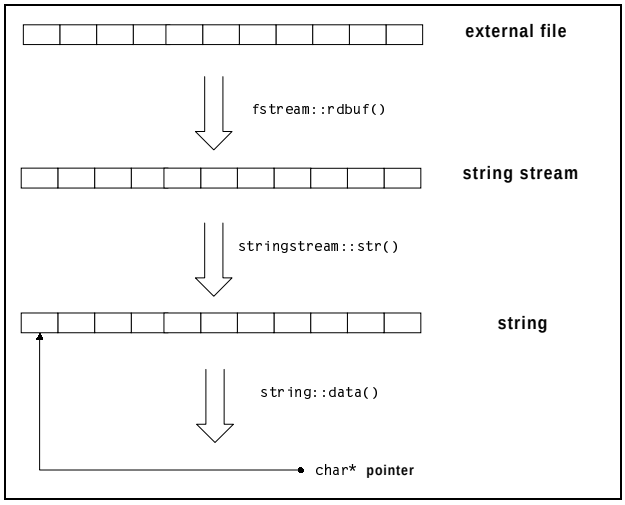


Figure 36. Copies of the file content

The content of the file is copied into the string stream's buffer when the pointer obtained through `rdbuf()` is inserted to the string stream. A second copy is created when the string stream's function `str()` is called.[36] The call to the string's function `data()` does not create yet another copy, but returns a pointer to the string's internal data.

2.10 Synchronizing Streams

In the previous section, we saw how streams can share stream buffers. In this section, we will see that streams can also share a file, as when streams in different processes exchange data through a file. Figure 37 graphically illustrates how streams share files:

[36] The traditional iostreams' `strstream` allows you to obtain a pointer to the stream's internal buffer. Different from the standard iostreams' `stringstream`, it does not create a copy of the internal data. Hence, using the deprecated `strstream` instead of the standard `stringstream` spares you the overhead of creating a second copy of the data.

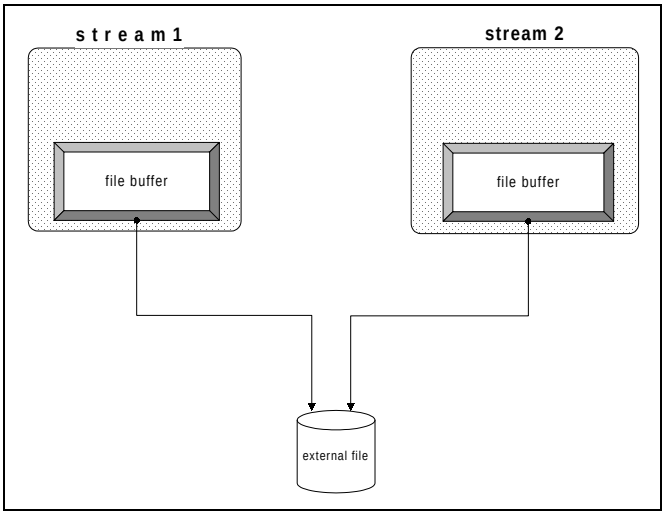


Figure 37. Streams sharing a file

Because streams use a buffer, the content of the file might be different from the content of the buffer that is supposed to reflect the file's content. When data is extracted through a file stream, a certain part of the file's content is read into the buffer; subsequent extractions access the buffer instead of the file. Once the file content is modified, the buffer content becomes obsolete. Similarly, when data is written through a file stream, the output is stored in the buffer and not written to the file. The file is accessed only when the buffer is full. For this reason, output from one stream will not be immediately available to the other stream.

2.10.1 Explicit Synchronization

You can force the stream to empty its buffer into an output file, or to refill its buffer from an input file. This is done through the stream buffer's function `pubsync()`. Typically, you will call `pubsync()` indirectly through functions of the stream layer. Input streams and output streams have different member functions that implicitly call `pubsync()`.

2.10.1.1 Output Streams

Output streams have a `flush()` function that writes the buffer content to the file. In case of failure, `badbit` will be set or an exception thrown, depending on the exception mask.

```
ofstream ofstr("/tmp/fil");
ofstr << "Hello ";                                    \\1
ofstr << "World!\n";
ofstr.flush();                                        \\2
```

`//1` The attempt to extract anything from the file `/tmp/fil` after this insertion will probably fail, because the string `"Hello "` is buffered and not yet written to the external file.

`//2` After the call to `flush()`, however, the file will contain "`Hello World!\n`". (Incidentally, the call to `ostr.flush()` can be replaced by the `flush` manipulator; i.e., `ostr << flush;`)

Keep in mind that flushing is a time-consuming operation. The function `flush()`not only writes the buffer content to the file; it may also reread the buffer in order to maintain the current file position. For the sake of performance, you should avoid inadvertent flushing, as when the `endl` manipulator calls `flush()` on inserting the end-of-line character. (See Section 2.8.1.2.)

2.10.1.2 Input Streams

Input streams have a `sync()` function. It forces the stream to access the external device and refill its buffer, beginning with the current file position.[37] The example below demonstrates the principle theoretically. In real-life, however, the two streams would belong to two separate processes and could use the shared file to communicate.

```
ofstream ofstr("/tmp/fil");
ifstream ifstr("/tmp/fil");
string s;

ofstr << "Hello "
ofstream::pos_type p = ofstr.tellp();
ofstr << "World!\n" << flush;
ifstr >> s;                                                    \\1

ofstr.seekp(p);
ofstr << "Peter!" << flush;                                    \\2
ifstr >> s;                                                    \\3

ofstr << " Happy Birthday!\n" << flush;                        \\4
ifstr >> s;                                                    \\5

ifstr.sync();                                                  \\6
ifstr >> s;
```

[37] In case of input streams the behavior of `sync()` is implementation-defined, i.e. is not standardized. The traditional iostreams had a `sync()` function that did the expected synchronization, i.e. refilling the buffer beginning with the current file position.

`//1` Here the input stream extracts the first string from the shared file. In doing so, the input stream fills its buffer. It reads as many characters from the external file as needed to fill the internal buffer. For this reason, the number of characters to be extracted from the file is implementation-specific; it depends on the size of the internal stream buffer.

`//2` The output stream overwrites part of the file content. Now the file content and the content of the input stream's buffer are inconsistent. The file contains `"Hello Peter!"`; the input stream's buffer still contains `"Hello World!"`.

`//3` This extraction takes the string `"World!"` from the buffer instead of yielding `"Peter!"`, which is the current file content.

`//4` More characters are appended to the external file. The file now contains `"Hello Peter! Happy Birthday!"`, whereas the input stream's buffer is still unchanged.

`//5` This extraction yields nothing. The input stream filled its buffer with the entire content of the file because the file is so small in our toy example. Subsequent extractions made the input stream hit the end of its buffer, which is regarded as the end of the file as well. The extraction results in `eofbit` set, and nothing will be extracted. There is no reason to ever access the external file again.

`//6` A call to `sync()` eventually forces the input stream to refill the buffer from the external device, beginning with the current file position. After the synchronization, the input stream's buffer will contain `"Happy Birthday!\n"`. The next extraction will yield `"Happy"`.

As the draft specifies the behavior of `sync()` as implementation-defined, you can alternatively try repositioning the input stream to the current position instead; i.e., `istr.seekg(ios_base::cur);`

Please note: **If you have to synchronize several streams that share a file, it is advisable to call the `sync()` function after each output operation and before each input operation.**

2.10.2 Implicit Synchronization Using the unitbuf Format Flag

You can achieve a kind of automatic synchronization for output files by using the format flag `ios_base::unitbuf`. It causes an output stream to flush its buffer after each output operation as follows:

```
ofstream ostr("/tmp/fil");
ifstream istr("/tmp/fil");

ostr << unitbuf;                                          \\1

while (some_condition)
```

```
{ ostr << "… some output…";                                    \\2
  // process the output
  istr >> s;
  // …
}
```

//1 Set the `unitbuf` format flag.

//2 After each insertion into the shared file `/tmp/fil`, the buffer is automatically flushed, and the output is available to other streams that read from the same file.

Since it is not overly efficient to flush after every single token that is inserted, you might consider switching off the `unitbuf` flag for a lengthy output that is not supposed to be read partially.

```
ostr.unsetf(ios_base::unitbuf);                                 \\1
ostr << " … some lengthy and complicated output …";
ostr.flush().setf(ios_base::unitbuf);                           \\2
```

//1 Switch off the `unitbuf` flag. Alternatively, using manipulators, you can say `ostr << nounitbuf;`

//2 Flush the buffer and switch on the `unitbuf` flag again. Alternatively, you can say `ostr << flush << unitbuf;`

2.10.3 Implicit Synchronization by Tying Streams

Another mechanism for automatic synchronization in certain cases is tying a stream to an output stream, as demonstrated in the code below. All input or output operations flush the tied stream's buffer before they perform the actual operation.

```
ofstream ostr("/tmp/fil");
ifstream istr("/tmp/fil");
ostream* old_tie = istr.tie(&ostr);                             //1

while (some_condition)
{ ostr << " some output ";
  string s;
  while (istr >> s)                                             //2
     // process input ;
}

istr.tie(old_tie);                                              //3
```

//1 The input stream `istr` is tied to the output stream `ostr`. The `tie()` function returns a pointer to the previously tied output stream, or zero if no output stream is tied.

//2 Before any input is done, the tied output stream's buffer is flushed so that the result of previous output operations to `ostr` is available in the external file `/tmp/fil`.

//3 The previous tie is restored.

2.10.4 Synchronizing the Predefined Standard Streams

The predefined streams `cin`, `cout`, `cerr`, and `clog` are examples of synchronized streams:

- `cin` is tied to `cout`; i.e., before each input operation on `cin`, the output stream `cout` is forced to flush its buffer.
- `cerr` is synchronized using the `unitbuf` format flag; i.e., after each output to `cerr`, its buffer is flushed.
- `clog` is connected to the same output channel and thus behaves like `cerr`, except that it is not synchronized with any of the other standard streams; i.e., it does not have the `unitbuf` flag set.

2.10.5 Synchronization with the C Standard I/O

The predefined C++ streams `cin`, `cout`, `cerr`, and `clog` are associated with the standard C files `stdin`, `stdout`, and `stderr`, as we saw in Section 2.3.1. This means that insertions into `cout`, for instance, go to the same file as output to `stdout`. By default, input and output to the predefined streams is synchronized with read or write operations on the standard C files. The effect is that input and output operations are executed in the order of invocation, independently of whether the operations used the predefined C++ streams or the standard C files.

This synchronization is time-consuming and thus might not be desirable in all situations. You can switch it off by calling:

```
sync_with_stdio(false);
```

After such a call, the predefined streams operate independently of the C standard files, with possible performance improvements in your C++ stream operations. However, you should call `sync_with_stdio()` prior to any input or output operation on the predefined streams, because otherwise the effect of calling `sync_with_stdio()` will be implementation-defined.

2.11 Stream Storage for Private Use: iword, pword, and xalloc

As we have seen in previous sections, a stream carries a number of data items: an error state, a format state, a locale, an exception mask, information about tied streams, and a stream buffer, to mention a few. Sometimes it is useful and necessary to store *additional* data in a stream.

2.11.1 An Example: Storing a Date Format String

Consider the inserter and extractor we defined for a date class in Section 2.7. The input and output operations were internationalized and relayed the task of date formatting and parsing to the stream's locale. Here, however, the rules for formatting and parsing were fixed, making them much more restricted than the features available in the standard C library, for example.

In the standard C library, you can specify format strings, similar to those for `prinft()` and `scanf()`, that describe the rules for parsing and formatting dates.[38] For example, the format string `"%A, %B %d, %Y"` stands for the rule

[38] See functions `strftime()`, `strptime()`, and `wcsftime()` in X/Open for reference.

that a date must consist of the name of the weekday, the name of the month, the day of the month, and the year—as in Friday, July 12, 1996.

Now imagine you want to improve the input and output operations for the date class by allowing specification of such format strings. How can you do this? Other format information is stored in the stream's format state; consequently, you may want to store the format string for dates somewhere in the stream as well. And indeed, you can.

Streams have an array for private use. An array element is of a `union` type that allows access as a `long` or as a pointer to `void`.[39] The array is of unspecified size, and new memory is allocated as needed. In principle, you can think of it as infinitely long.

You can use this array to store in a stream whatever additional information you might need. In our example, we would want to store the format string.

The array can be accessed by two functions: `iword()` and `pword()`. Both functions take an index to an array element and return a reference to the respective element. The function `iword()` returns a reference to `long`; the function `pword()` allows access to the array element as a pointer to `void`.

Indices into the array are maintained by the `xalloc()` function, a static function in class `ios_base` that returns the next free index into the array.

2.11.2 Another Look at the Date Format String

We would like to store the format string for dates in the iostream storage through `iword()` and `pword()`. In this way, the input and output operations for `date` objects can access the format string for parsing and formatting. Format parameters are often set with manipulators (see Section 2.3.3.2), so we should add a manipulator that sets the date format string. It could be used like this:

```
date today;
ofstream ostr;
// …
ostr << setfmt("%D") << today;
```

Here is a suggested implementation for such a manipulator:

```
class setfmt : public smanip<const char*>
{
 public:
  setfmt(const char* fmt)
  : smanip<const char*>(setfmt_,fmt) {}
 private:
  static const int datfmtIdx ;                                \\1
```

[39] According to the draft working paper, they are two separate arrays. However, RW's implementation uses the old technique involving only one array.

```
    static ios_base& setfmt_(ios_base& str, const char* fmt)
    {
      str.pword(datfmtIdx) = (void*) fmt;                        \\2
      return str;
    }

    template<class charT, class Traits>
    friend basic_ostream<charT, Traits> &                        \\3
    operator << (
    basic_ostream<charT, Traits >& os, const date& dat);
  };

  const int setfmt::datfmtIdx = ios_base::xalloc();              \\4
```

The technique applied to implement the manipulator is described in detail in Example 2 of Section 2.8.2.3, so we won't repeat it here. But regarding this manipulator and the private use of iostream storage , there are other interesting details:

`//1` The manipulator class owns the index of the element in the iostream storage where we want to store the format string. It is initialized in `\\4` by a call to `xalloc()`.

`//2` The manipulator accesses the array `pword()` using the index `datfmtIdx`, and stores the pointer to the date format string.[40] Note that the reference returned by `pword()` is only used for *storing* the pointer to the date format string. Generally, one should never store a reference returned by `iword()` or `pword()` in order to access the stored data through this reference later on. This is because these references can become invalid once the array is reallocated or copied. (See the *Class Reference* for more details.)

`//3` The inserter for `date` objects needs to access the index into the array of pointers, so that it can read the format string and use it. Therefore, the inserter has to be declared as a friend. In principle, the extractor would have to be a friend, too; however, the standard C++ locale falls short of supporting the use of format strings like the ones used by the standard C function `strptime()`. Hence, the implementation of a date extractor that supports date format strings would be a lot more complicated than the implementation for the inserter, which can use the stream's locale. We have omitted the extractor for the sake of brevity.

The inserter for `date` objects given below is almost identical to the one we described in Section 2.7.3:

```
template<class charT, class Traits>
basic_ostream<charT, Traits> &
operator << (basic_ostream<charT, Traits >& os, const date& dat)
```

---

[40] Error handling is omitted in the example because the standard does not indicate how `pword()` and `iword()` indicate failure. Possible choices would be to throw a `bad_alloc` exception or to set the `failbit`.

```
{
 ios_base::iostate err = 0;
 char*  patt = 0;
 int    len  = 0;
 charT* fmt  = 0;

 try {
  typename basic_ostream<charT, Traits>::sentry opfx(os);

  if(opfx)
  {
     patt = (char*) os.pword(setfmt.datfmtIdx);              \\1
     len  = strlen(patt);
     fmt  = new charT[len];

     use_facet<ctype<charT> >(os.getloc()).
         widen(patt, patt+len, fmt);

     if (use_facet<time_put<charT
                  ,ostreambuf_iterator<charT,Traits> > >
          (os.getloc())
          .put(os,os,os.fill(),&dat.tm_date,fmt,fmt+len)     \\2
          .failed()
        )
          err = ios_base::badbit;
     os.width(0);
  }
 } //try
 catch(...)
 {
    delete [] fmt;
    bool flag = FALSE;
    try {
          os.setstate(ios_base::failbit);
        }
    catch( ios_base::failure ) { flag= TRUE; }
    if ( flag ) throw;
 }

 delete [] fmt;
 if ( err ) os.setstate(err);

 return os;
}
```

The only change from the previous inserter is that the format string here is read from the iostream storage (in statement `//1`) instead of being the fixed string `"%x"`. The format string is then provided to the locale's time formatting facet (in statement `//2`).

2.11.3 Caveat

Note that the solution suggested here has a pitfall.

The manipulator takes the format specification and stores it. The inserter retrieves it and uses it. In such a situation, the question arises: Who owns the format string? In other words, who is responsible for creating and deleting it and hence controlling its lifetime? Neither the manipulator nor the inserter can own it because they share it.

We solved the problem by requiring the format specification to be created and deleted by the iostream user. Only a pointer to the format string is handed over to the manipulator, and only this pointer is stored through `pword()`. Also, we do not copy the format string because it would not be clear who—the manipulator or the inserter—is responsible for deleting the copy. Hence the iostream user has to monitor the format string's lifetime, and ensure that the format string is valid for as long as it is accessed by the inserter.

This introduces a subtle lifetime problem in cases where the date format is a variable instead of a constant: the stream might be a static stream and hence live longer that the date format variable. This is a problem you will always deal with when storing a pointer or reference to additional data instead of copying the data.

However, this subtle problem does not impose an undue burden on the user of our `setfmt` manipulator. If a static character sequence is provided, as in:

```
cout << setfmt("%A, %B %d, %Y") << today;
```

the `setfmt` manipulator can be used safely, even with static streams like `cout`.

2.12 Creating New Stream Classes by Derivation

Sometimes it is useful to derive a stream type from the standard iostreams. This is the case when you want to add data members or functions, or modify the behavior of a stream's I/O operations.

In Section 2.11, we learned that additional data can be added to a stream object by using `xalloc()`, `iword()`, and `pword()`. However, this solution has a certain weakness in that only a pointer to the additional data can be stored and someone else has to worry about the actual memory.

This weakness can be overcome by deriving a new stream type that stores the additional data as a data member. Let's consider again the example of the `date` inserter and the `setfmt` manipulator from Section 2.11. Here let's derive a new stream that has an additional data member for storing the format string together with a corresponding member function for setting the date format specification.[41] Again, we will confine the example to the inserter of the `date` object and omit the extractor. Instead of inserting into an output stream, as we did before, we will now use a new type of stream called `odatstream`:

```
date today;
```

---

[41] This, of course, is a toy example. You would probably never derive a new class for adding only one data member. However, it keeps the example simple and allows us to demonstrate the principle of deriving new stream classes.

```
odatstream ostr(cout);
// …
ostr << setfmt("%D") << today;
```

In the next sections, we will explore how we can implement such a derived stream type.

2.12.1 Choosing a Base Class

The first question is: Which of the standard stream classes shall be the base class? The answer fully depends on the kind of addition or modification you want to make. In our case, we want to add formatting information, which depends on the stream's character type since the format string is a sequence of tiny characters. As we will see later on, the format string must be expanded into a sequence of the stream's character type for use with the stream's locale. Consequently, a good choice for a base class is class `basic_iostream <charT,Traits>`, and since we want the format string to impact only output operations, the best choice is class `basic_ostream <charT,Traits>`.

In general, you choose a base class by looking at the kind of addition or modification you want to make and comparing it with the characteristics of the stream classes.

- Choose `ios_base` if you add information and services that do not depend on the stream's character type.
- Choose `basic_ios<charT, Traits>` if the added information does depend on the character type, or requires other information not available in `ios_base`, such as the stream buffer.
- Derive from the stream classes `basic_istream <charT,Traits>`, `basic_ostream <charT,Traits>`, or `basic_iostream <charT, Traits>` if you want to add or change the input and output operations.
- Derive from the stream classes `basic_(i/o)fstream <charT,Traits>`, or `basic_(i/o)stringstream <charT, Traits, Allocator>` if you want to add or modify behavior that is file- or string-related, such as the way a file is opened.

Derivations from `basic_istream <charT,Traits>`, `basic_ostream <charT,Traits>`, or `basic_iostream <charT, Traits>` are the most common cases, because you typically want to modify or add input and output operations.

If you derive from `ios_base` or `basic_ios<charT, Traits>`, you do not inherit any input and output operations; you do this if you really want to reimplement all of them or intend to implement a completely different kind of input or output operation, such as unformatted binary input and output.

Derivations from file or string streams such as `basic_(i/o)fstream <charT,Traits>` or `basic_(i/o)stringstream <charT, Traits, Allocator>`

are equally rare, because they make sense only if file- or string-related data or services must be added or modified.

Choose `basic_istream <charT,Traits>`, `basic_ostream <charT,Traits>`, or `basic_iostream <charT, Traits>` as a base class when deriving new stream classes, unless you have good reason not to do so.

2.12.2 Construction and Initialization

All standard stream classes have class `basic_ios<charT,Traits>` as a virtual base class. In C++, a virtual base class is initialized by its most derived class; i.e., our new `odatstream` class is responsible for initialization of its base class `basic_ios<charT,Traits>`. Now class `basic_ios<charT,Traits>` has only one public constructor, which takes a pointer to a stream buffer. This is because class `basic_ios<charT,Traits>` contains a pointer to the stream buffer, which has to be initialized when a `basic_ios` object is constructed. Consequently, we have to figure out how to provide a stream buffer to our base class. Let's consider two options:

- Derivation from file stream or string stream classes; i.e., class `(i/o)fstream<>` or class `(i/o)stringstream<>`, and
- Derivation from the stream classes `basic_(i/o)stream<>`.

2.12.2.1 Derivation from File Stream or String Stream Classes Like (i/o)fstream<> or (i/o)stringstream<>

The file and string stream classes contain a stream buffer data member and already monitor their virtual base class's initialization by providing the pointer to their own stream buffer. If we derive from one of these classes, we will not provide another stream buffer pointer because it would be overwritten by the file or string stream's constructor anyway. (Remember that virtual base classes are constructed before non-virtual base classes regardless of where they appear in the hierarchy.) Consider:

```
template <class charT, class Traits=char_traits<charT> >
class MyOfstream : public basic_ofstream<charT,Traits> {
 public:
   MyOfstream(const char* name)
   : basic_ios<charT,Traits>(…streambufptr…)
   , basic_ofstream<charT,Traits>(name) {}
   // . . .
};
```

The order of construction would be:

```
basic_ios(basic_streambuf<charT,Traits>*)
basic_ofstream(const char*)
basic_ostream(basic_streambuf<charT,Traits>*)
ios_base()
```

In other words, the constructor of basic_ofstream overwrites the stream buffer pointer set by the constructor of basic_ios.

To avoid this dilemma, class basic_ios<charT,Traits> has a protected default constructor in addition to its public constructor. This default constructor, which requires a stream buffer pointer, doesn't do anything. Instead, there is a protected initialization function basic_ios<charT,Traits>::init() that can be called by any class derived from basic_ios<charT,Traits>. With this function, initialization of the basic_ios<> base class is handled by the stream class that actually provides the stream buffer—in our example, basic_ofstream<charT,Traits>. It will call the protected init() function:

```
template <class charT, class Traits=char_traits<charT> >
class MyOfstream : public basic_ofstream<charT,Traits> {
 public:
   MyOfstream(const char* name)
   : basic_ofstream<charT,Traits>(name) {}
   // . . .
};
```

The order of construction and initialization is:

```
basic_ios()
basic_ofstream(const char*)
basic_ostream()
```
which calls:
```
  basic_ios<charT,Traits>::init(basic_streambuf<charT,Traits>*)
ios_base()
```

2.12.2.2 Derivation from the Stream Classes basic_(i/o)stream<>

The scheme for deriving from the stream classes is slightly different in that you must always provide a pointer to a stream buffer. This is because the stream classes do not contain a stream buffer, as the file or string stream classes do. For example, a class derived from an output stream could look like this:

```
template <class charT, class Traits=char_traits<charT> >
class MyOstream : public basic_ostream<charT,Traits> {
 public:
   MyOstream(basic_streambuf<charT,Traits>* sb)
   : basic_ostream<charT,Traits>(sb) {}
   // . . .
};
```

There are several ways to provide the stream buffer required for constructing such a stream:

- **Create the stream buffer independently, before the stream is created.** Here is a simple example in which a file buffer is created as a separate object and used by the derived stream:

```
basic_filebuf<char> strbuf;
strbuf.open("/tmp/xxx");
MyOstream<char> mostr(&strbuf);
mostr << "Hello world\n";
```

- **Take the stream buffer from another stream.** In the example below, the stream buffer is “borrowed” from the standard error stream `cerr`:

  ```
  MyOstream<char,char_traits<char> > mostr(cerr.rdbuf());
  mostr << "Hello world\n";
  ```

  Remember that the stream buffer is now shared between `mostr` and `cerr` (see Section 2.9.2 for details).

- **Contain the stream buffer in the derived stream, either as a data member or inherited**. This option is typically preferred when a new stream buffer type is used along with the new stream type.

2.12.3 The Example

Let’s return now to our example, in which we are creating a new stream class by derivation.

2.12.3.1 The Derived Stream Class

Let us derive a new stream type `odatstream` that has an additional data member `fmt_` for storing a date format string, together with a corresponding member function `fmt()` for setting the date format specification.

```
template <class charT, class Traits=char_traits<charT> >
class odatstream : public basic_ostream <charT,Traits>
{l
public:
 odatstream(basic_ostream<charT,Traits>& ostr,const char* fmt =
"%x")                                                          \\1
 : basic_ostream<charT,Traits>(ostr.rdbuf())
 {
   fmt_=new charT[strlen(fmt)];
   use_facet<ctype<charT> >(ostr.getloc()).
       widen(fmt, fmt+strlen(fmt), fmt_);                      \\2
 }

 basic_ostream<charT,Traits>& fmt(const char* f)              \\3
 {
   delete[] fmt_;
   fmt_=new charT[strlen(f)];
   use_facet<ctype<charT> >(os.getloc()).
       widen(f, f+strlen(f), fmt_);
   return *this;
 }

 charT const* fmt() const                                    \\4
 {
  charT * p = new charT[Traits::length(fmt_)];
  Traits::copy(p,fmt_,Traits::length(fmt_));
  return p;
 }
 ~odatstream()                                                \\5
 { delete[] fmt_; }


private:
 charT* fmt_;                                                 \\6
```

```
  template <class charT, class Traits>                          \\7
  friend basic_ostream<charT, Traits> &
  operator << (basic_ostream<charT, Traits >& os, const date& dat);
 };
```

//1 A date output stream borrows the stream buffer of an already existing output stream, so that the two streams will share the stream buffer.

The constructor also takes an optional argument, the date format string. This is always a sequence of tiny characters.

//2 The format string is widened or translated into the stream's character type `charT`. This is because the format string will be provided to the time facet of the stream's locale, which expects an array of characters of type `charT`.

//3 This version of function `fmt()` allows you to set the format string.

//4 This version of function `fmt()` returns the current format string setting.

//5 The date stream class needs a destructor that deletes the format string.

//6 A pointer to the date format specification is stored as a private data member `fmt_`.

//7 The inserter for dates will have to access the date format specification. For this reason, we make it a friend of class `odatstream`.

2.12.3.2 The Date Inserter

We would like to be able to insert `date` objects into all kinds of output streams. And, whenever the output stream is a date output stream of type `odatstream`, we would like to take advantage of its ability to carry additional information for formatting date output. How can this be achieved?

It would be ideal if the inserter for `date` objects were a virtual member function of all output stream classes that we could implement differently for different types of output streams. For example, when a date object is inserted into an `odatstream`, the formatting would use the available date formatting string; when inserted into an arbitrary output stream, default formatting would be performed. Unfortunately, we cannot modify the existing output stream classes, since they are part of a library you will not want to modify.

This kind of problem is typically solved using dynamic casts. Since the stream classes have a virtual destructor, inherited from class `basic_ios`, we can use dynamic casts to achieve the desired virtual behavior.[42]

Here is the implementation of the date inserter, which is similar to the one in Section 2.7.2. The differences are shaded:

```
template<class charT, class Traits>
basic_ostream<charT, Traits> &
operator << (basic_ostream<charT, Traits >& os, const date& dat)
{
 ios_base::iostate err = 0;

 try {
  typename basic_ostream<charT, Traits>::sentry opfx(os);

  if(opfx)
  {
    charT* fmt;
    charT buf[3];

    try {                                                    \\1
      odatstream<charT,Traits>*
      p = dynamic_cast<odatstream<charT,Traits>*>(&os);      \\2
    }
    catch (bad_cast)                                         \\3
    {
      char patt[3] = "%x";

      use_facet(os.getloc(),
        (ctype<charT>*)0).widen(patt,patt+3,buf);
    }
```

---

[42] For a more detailed discussion of the problem and its solution, see Section 14.2, p. 306ff, of Bjarne Stroustrup, "The Design and Evolution of C++," Addison-Wesley 1994.

```
    fmt = (p) ? p->fmt_ : buf;                                  \\4

    if (use_facet<time_put<charT,ostreambuf_iterator<charT,Traits>
> >(os.getloc())

.put(os,os,os.fill(),&dat.tm_date,fmt,fmt+Traits::length(fmt)).fai
led())
       err  = ios_base::badbit;
    os.width(0);
  }
 } //try
 catch(...)
 {
    bool flag = FALSE;
    try {
           os.setstate(ios_base::failbit);
     }
    catch( ios_base::failure ) { flag= TRUE; }
    if ( flag ) throw;
 }

 if ( err ) os.setstate(err);

 return os;
}
```

`//1` We will perform a dynamic cast in statement `//2`. A dynamic cast throws an exception in case of mismatch. Naturally, we do not want to confront our user with `bad_cast` exceptions because the mismatch does not signify an error condition, but only that the default formatting will be performed. For this reason, we will try to catch the potential `bad_cast` exception.

`//2` This is the dynamic cast to find out whether the stream is a date stream or any other kind of output stream.

`//3` In case of mismatch, we prepare the default date format specification `"%x"`.

`//4` If the stream is not of type `odatstream`, the default format specification prepared in the catch clause is used. Otherwise, the format specification is taken from the private data member `fmt_`.

2.12.3.3 The Manipulator

The date output stream has a member function for setting the format specification. Analogous to the standard stream format functions, we would like to provide a manipulator for setting the format specification. This manipulator will affect only output streams. Therefore, we have to define a manipulator base class for output stream manipulators, `osmanip`, along with the necessary inserter for this manipulator. We do this in the code below. See Section 2.8 for a detailed discussion of the technique we are using here:

```
template <class Ostream, class Arg>
class osmanip {
  public:
    typedef Ostream ostream_type;
    typedef Arg argument_type;
```

```
    osmanip(Ostream& (*pf)(Ostream&, Arg), Arg arg)
    : pf_(pf) , arg_(arg) { ; }

  protected:
    Ostream&     (*pf_)(Ostream&, Arg);
    Arg          arg_;

  friend Ostream&
    operator<< (Ostream& ostr, const osmanip<Ostream,Arg>& manip);
};

template <class Ostream, class Arg>
Ostream& operator<< (Ostream& ostr,const osmanip<Ostream,Arg>&
manip)
{
   (*manip.pf_)(ostr,manip.arg_);
   return ostr;
}
```

After these preliminaries, we can now implement the `setfmt` manipulator itself:

```
template <class charT, class Traits>
inline basic_ostream<charT,Traits>&
sfmt(basic_ostream<charT,Traits>& ostr, const char* f)        \\1
{
 try {                                                        \\2
 odatstream<charT,Traits>* p =
dynamic_cast<odatstream<charT,Traits>*>(&ostr);
}
catch (bad_cast)                                              \\3
{ return ostr; }

 p->fmt(f);                                                   \\4
 return ostr;
}

template <class charT,class Traits>
inline osmanip<basic_ostream<charT,Traits>,const char*>
setfmt(const char* fmt)
{ return osmanip<basic_ostream<charT,Traits>,const
char*>(sfmt,fmt); }                                           \\5
```

`//1` The function `sfmt()` is the function associated with the `setfmt` manipulator. Its task is to take a format specification and hand it over to the stream. This happens only if the stream is a date output stream; otherwise, nothing is done.

`//2` We determine the stream's type through a dynamic cast. As it would be rather drastic to let a manipulator call result in an exception thrown, we catch the potential `bad_cast` exception.

`//3` In case of mismatch, we don't do anything and simply return.

`//4` In case the stream actually is a date output stream, we store the format specification by calling the stream's `fmt()` function.

`//5` The manipulator itself is a function that creates an output manipulator object.

2.12.3.4 A Remark on Performance

The solution suggested in the previous Sections 2.12.3.2 and 2.12.3.3 uses dynamic casts and exception handling to implement the date inserter and the date format manipulator. Although this technique is elegant and makes proper use of the C++ language, it might introduce some loss in runtime performance due to the use of exception handling. This is particularly true as the dynamic cast expression, and the exception it raises, is used as a sort of branching statement. In other words, the "exceptional" case occurs relatively often and is not really an exception.

If optimal performance is important, you can choose an alternative approach: in the proposed solution that uses dynamic casts, extend the date inserter for arbitrary output streams `basic_ostream<charT,Traits>& operator<< (basic_ostream <charT,Traits>&, const date&)` so that it formats dates differently, depending on the type of output stream. Alternatively, you can leave the existing date inserter for output streams unchanged and implement an additional date inserter that works for output *date* streams only; its signature would be `odatstream<charT,Traits>& operator<< (odatstream<charT,Traits>&, const date&)`. Also, you would have two manipulator functions, one for arbitrary output streams and one for output date streams only, that is, `basic_ostream<charT,Traits>& sfmt (basic_ostream<charT,Traits>&, const char*)` and `odatstream<charT,Traits>& sfmt (odatstream<charT,Traits>&, const char*)`. In each of the functions for date streams, you would replace those operations that are specific for output *date* streams.

This technique has the drawback of duplicating most of the inserter's code, which in turn might introduce maintenance problems. The advantage is that the runtime performance is likely to be improved.

2.12.4 Using iword/pword for RTTI in Derived Streams

In the previous section, we discussed an example that used runtime-type identification (RTTI) to enable a given input or output operation to adapt its behavior based on the respective stream type's properties.

Before RTTI was introduced into the C++ language in the form of the new style casts `dynamic_cast<>`, the problem was solved using `iword()`, `pword()`, and `xalloc()` as substitutes for runtime-type identification (RTTI).[43] We describe this old-fashioned technique only briefly because, as the previous example suggests, the use of dynamic casts is clearly preferable over the

---

[43] An introduction to this technique can also be found in Section 6.10, p. 90ff, of Steve Teale's "C++ IOStreams Handbook," Addison-Wesley 1993. Watch out, the example given there has several severe bugs!

RTTI substitute. Still, the traditional technique might be useful if your current compiler does not yet support the new-style casts.

The basic idea of the traditional technique is that the stream class and all functions and classes that need the runtime type information, like the inserter and the manipulator function, agree on two things:

- An index into the arrays for additional storage; in other words, *Where* do I find the RTTI?, and
- The content or type identification that all concerned parties expect to find there; in other words, *What* will I find?

In the sketch below, the derived stream class reserves an index into the additional storage. The index is a static data member of the derived stream class, and identifies all objects of that stream class. The content of that particular slot in the stream's additional storage, which is accessible through `pword()`, is expected to be the respective stream object's `this` pointer.

Here are the modifications to the derived class `odatstream`:

```
template <class charT, class Traits=char_traits<charT> >
class odatstream : public basic_ostream <charT,Traits>
{
public:
 static int xindex()                                          \\1
 {
  static int inited = 0;
  static int value = 0;
  if (!inited)
  {
     value = xalloc();
     inited++;
  }
  return value;
 }

 odatstream(basic_ostream<charT,Traits>& ostr,const char* fmt =
"%x")
 : basic_ostream<charT,Traits>(ostr.rdbuf())
 {
   pword(xindex()) = this;                                    \\2

   fmt_=new charT[strlen(fmt)];
   use_facet<ctype<charT> >(ostr.getloc()).widen(fmt,
fmt+strlen(fmt), fmt_);
 }

 // … other member, as in the previous section …
};
```

`//1` The static function `xindex()` is concerned with reserving the index into the arrays for additional storage. It also serves as the access function to the index.

`//2` The reserved slot in the arrays for additional storage is filled with the object's own address.

Here are the corresponding modifications to the manipulator:

```
template <class charT, class Traits>
inline basic_ostream<charT,Traits>&
sfmt(basic_ostream<charT,Traits>& ostr, const char* f)
{
 if (ostr.pword(odatstream<charT,Traits>::xindex()) == &ostr) \\1
    ((odatstream<charT,Traits>&)ostr).fmt(f);
 return ostr;
}
```

`//1` The manipulator function checks whether the content of the reserved slot in the stream's storage is the stream's address. If it is, the stream is considered to be a date output stream.

Note that the technique described in this section is not safe. There is no way to ensure that date output streams and their related functions and classes are the only ones that access the reserved slot in a date output stream's additional storage. In principle, every stream object of any type can access all entries through `iword()` or `pword()`. It's up to your programming discipline to restrict access to the desired functions. It is unlikely, however, that all streams will make the same assumptions about the storage's content. Instead of agreeing on each stream object's address as the run-time-type identification, we also could have stored certain integers, pointers to certain strings, etc. Remember, it's the combination of reserved index *and* assumed content that represents the RTTI substitute.

2.13 Defining A Code Conversion Facet

File stream buffers are responsible for the transport of characters to and from an external device. In many cases, the character encoding used internally inside your program and externally on the device will differ. Hence the file stream buffer will have to convert characters from one encoding to another each time it reads from or writes to the external device. (This *User's Guide* section on internationalization gives a detailed discussion of character encodings and explains a couple of typical code conversions. If you are not familiar with code conversions, we recommend you read about them before delving into the details of implementing one, which will be explained in this section.)

A code conversion is not performed by the file stream buffer itself. This task is encapsulated in a code conversion facet. Each time the file stream buffer has to convert characters, it consults its locale's code conversion facet for the actual conversion. For this reason, file stream buffers and code conversion facets have to work together closely, and the file stream buffer depends on its locale's code conversion facet.

This clear separation of responsibilities enables you to change a file stream's behavior substantially, without touching the file stream class itself. All you have to do is provide a special code conversion facet. In doing so, you turn an ordinary file stream into one that converts, say, EBCDIC files on a mainframe's file system into a stream of ASCII characters for internal processing.

However, the task of implementing a code conversion facet requires a thorough understanding of the way file stream buffers and code conversion facets interact. In this section, we will use two examples to explain the principles of this interaction.

Before we move on to the examples, let's go through an overview of the different kinds of code conversions. As we will see later on, different types of code conversions require different kinds of implementations.

2.13.1 Categories of Code Conversions

Code conversions fall into various categories depending on the properties of the character encodings involved. There are:

- Constant-size conversions, and
- Multibyte conversions, which again fall into the categories of:
    - State-independent conversions, and
    - State-dependent conversions.

Constant-size conversions are between character encodings where all characters are of equal size. All single- or wide-character encodings are examples of such character encodings. Each single character stands for itself and can be recognized and translated independently of its context. Conversions between ASCII and EBCDIC, or Unicode and ISO10646, are examples of constant-size conversions.

Multibyte conversions involve multibyte encodings. In multibyte encodings, characters have varying size. Some multibyte characters consist of two or more bytes, while others are represented by just one byte.

There is a substantial difference between code conversions involving state-dependent character encodings, and conversions between state-independent encodings. (Again, see this *User's Guide* section on internationalization for further details.)

State-dependent multibyte conversions involve one character encoding that is state-dependent. In state-dependent character encodings, character sequences can have different meanings depending on the current context. State-dependent encodings typically have *modes* and escape sequences that allow switching between modes. An example of a state-dependent character conversion is the conversion between the state-dependent JIS encoding for Japanese characters and the Unicode wide-character encoding.

State-independent multibyte conversions do not have modes. A sequence of characters can always be interpreted independently of its context. An example of a state-independent multibyte conversion is the conversion between EUC, which a state-independent multibyte encoding, and Unicode.

2.13.2 Example 1: Defining a Tiny Character Code Conversion (ASCII <-> EBCDIC)

As an example of how file stream buffers and code conversion facets collaborate, we would now like to implement a code conversion facet that can translate text files encoded in EBCDIC into character streams encoded in ASCII. The conversion between ASCII characters and EBCDIC characters is a constant-size code conversion where each character is represented by one byte. Hence the conversion can be done on a character-by-character basis.

To implement and use an ASCII-EBCDIC code conversion facet, we will:

1. Derive a new facet type from the standard code conversion facet type `codecvt`.
2. Specialize the new facet type for the character type `char`.
3. Implement the member functions that are used by the file buffer.
4. Imbue a file stream's buffer with a locale that carries an ASCII-EBCDIC code conversion facet.

The following sections will explain these steps in detail.

2.13.2.1 Derive a New Facet Type

Here is the new code conversion facet type `AsciiEbcdicConversion`:

```
template <class internT, class externT, class stateT>
class AsciiEbcdicConversion
: public codecvt<internT, externT, stateT>
{
};
```

It is empty because we will specialize the class template for the character type `char`.

2.13.2.2 Specialize the New Facet Type and Implement the Member Functions

Each code conversion facet has two main member functions, `in()` and `out()`:

- Function `in()`is responsible for the conversion done on reading from the external device; and
- Function `out()`is responsible for the conversion necessary for writing to the external device.

The other member functions of a code conversion facet used by a file stream buffer are:

- The function `always_noconv()`, which returns `true` if no conversion is performed by the facet. This is because file stream buffers might want to bypass the code conversion facet when no conversion is necessary; e.g.,

when the external encoding is identical to the internal. Our facet obviously will perform a conversion and does not want to be bypassed, so `always_noconv()` will return `false` in our example.

- The function `encoding()`, which provides information about the type of conversion; i.e., whether it is state-dependent or constant-size, etc. In our example, the conversion is constant-size. The function `encoding()` is supposed to return the size of the internal characters, which is 1 because the file buffer uses an ASCII encoding internally.

All public member functions of a facet call the respective, protected virtual member function, named `do_...()`. Here is the declaration of the specialized facet type:

```
class AsciiEbcdicConversion<char, char, mbstate_t>
: public codecvt<char, char, mbstate_t>
{
protected:

 result do_in(mbstate_t& state
              ,const char* from, const char* from_end, const
char*& from_next
              ,char* to        , char* to_limit      , char*&
to_next) const;

 result do_out(mbstate_t& state
              ,const char* from, const char* from_end, const
char*& from_next
              ,char* to        , char* to_limit      , char*&
to_next) const;

 bool do_always_noconv() const thow()
 { return false; };

 int do_encoding() const throw();
 { return  1; }

};
```

For the sake of brevity, we implement only those functions used by Rogue Wave's implementation of file stream buffers. If you want to provide a code conversion facet that is more widely usable, you would also have to implement the functions `do_length()` and `do_max_length()`.

The implementation of the functions `do_in()` and `do_out()` is straightforward. Each of the functions translates a sequence of characters in the range `[from,from_end)` into the corresponding sequence `[to,to_end)`. The pointers `from_next` and `to_next` point one beyond the last character successfully converted. In principle, you can do whatever you want, or whatever it takes, in these functions. However, for effective communication with the file stream buffer, it is important to indicate success or failure properly.

2.13.2.3 Use the New Code Conversion Facet

Here is an example of how the new code conversion facet can be used:

```
fstream inout("/tmp/fil");                                    \\1
AsciiEbcdicConversion<char,char,mbstate_t> cvtfac;
locale cvtloc(locale(),&cvtfac);
inout.rdbuf()->pubimbue(cvtloc)                               \\2
cout << inout.rdbuf();                                        \\3
```

`//1` When a file is created, a snapshot of the current global locale is attached as the default locale. Remember that a stream has two locale objects: one used for formatting numeric items, and a second used by the stream's buffer for code conversions.

`//2` Here the stream buffer's locale is replaced by a copy of the global locale that has an ASCII-EBCDIC code conversion facet.

`//3` The content of the EBCDIC file `"/tmp/fil"` is read, automatically converted to ASCII, and written to `cout`.

2.13.3 Error Indication in Code Conversion Facets

Since file stream buffers depend on their locale's code conversion facet, it is important to understand how they communicate. On writing to the external device, the file stream buffer hands over the content of its internal character buffer, partially or entirely, to the code conversion facet; i.e., to its `out()` function. It expects to receive a converted character sequence that it can write to the external device. The reverse takes place, using the `in()` function, on reading from the external file.

In order to make the file stream buffer and the code conversion facet work together effectively, it is necessary that the two main functions `in()` and `out()` indicate error situations the way the file stream buffer expects them to do it.

There are four possible return codes for the functions `in()` and `out()`:

- `ok`, which should obviously be returned when the conversion went fine.
- `partial`, which should be returned when the code conversion reaches the end of the input sequence `[from,from_end)` before a new character can be created. The file stream buffer's reaction to `partial` is to provide more characters and call the code conversion facet again, in order to successfully complete the conversion.[44]

---

[44] In our example of a conversion between ASCII and EBCDIC, we have no reason to ever return `partial`, because this is a conversion of single byte characters. Either a character can be recognized and converted, or the conversion fails; that is, `error` will be returned. The `partial` return code only makes sense in wide-character and multibyte conversions.

- `error`, which indicates a violation of the conversion rules; i.e., the character sequence to be converted does not obey the expected rules and thus cannot be recognized and converted. In this situation, the file stream buffer stops doing anything, and the file stream eventually sets its state to `badbit` and throws an exception if appropriate.
- `noconv`, which is returned if no conversion was needed.

2.13.4 Example 2: Defining a Multibyte Character Code Conversion (JIS <-> Unicode)

Let us consider the example of a state-dependent code conversion. As mentioned previously, this type of conversion would occur between JIS, which is a state-dependent multibyte encoding for Japanese characters, and Unicode, which is a wide-character encoding. As usual, we assume that the external device uses multibyte encoding, and the internal processing uses wide-character encoding.

Here is what you have to do to implement and use a state-dependent code conversion facet:

1. Define a new conversion state type if necessary.
2. Define a new character traits type if necessary, or instantiate the character traits template with the new state type.
3. Define the code conversion facet.
4. Instantiate new stream types using the new character traits type.
5. Imbue a file stream's buffer with a locale that carries the new code conversion facet.

These steps are explained in detail in the following sections.

2.13.4.1 Define a New Conversion State Type

While parsing or creating a sequence of multibytes in a state-dependent multibyte encoding, the code conversion facet has to maintain a conversion state. This state is by default of type `mbstate_t`, which is the implementation-dependent state type defined by the C library. If this type does not suffice to keep track of the conversion state, you have to provide your own conversion state type. We will see how this is done in the code below, but please note first that the new state type must have the following member functions:

- A constructor. The argument `0` has the special meaning of creating a conversion state object that represents the initial conversion state;
- Copy constructor and assignment;
- Comparison for equality and inequality.

Now here is the sketch of a new conversion state type:

```
class JISstate_t {
public:
                    JISstate_t( int state=0 )
                    : state_(state) { ; }

                    JISstate_t(const JISstate_t& state)
                    : state_(state.state_) { ; }

                    JISstate_t& operator=(const JISstate_t& state)
                     {
                        if ( &state != this )
                          state_= state.state_;
                        return *this;
                     }

                    JISstate_t& operator=(const int state)
                     {
                        state_= state;
                        return *this;
                     }

                    bool operator==(const JISstate_t& state) const
                     {
                        return ( state_ == state.state_ );
                     }

                    bool operator!=(const JISstate_t& state) const
                     {
                        return ( !(state_ == state.state_) );
                     }

private:
                    int state_;

                  };
```

2.13.4.2 Define a New Character Traits Type

The conversion state type is part of the character traits. Hence, with a new conversion state type, you need a new character traits type.

Rogue Wave's implementation of the *Standard C++ Library* has a non-standard extension to the standard character traits class template `char_traits`. The extension is an additional template parameter for the conversion state type. For this reason, you can create a new character traits type by instantiating the character traits with your new conversion state type:

```
char_traits<wchar_t, JISstate_t>
```

However, if you do not want to rely on a non-standard and thus non-portable feature of the library, you have to define a new character traits type and redefine the necessary types:

```
struct JIS_char_traits: public char_traits<wchar_t>
{
        typedef JISstate_t                state_type;
        typedef fpos<state_type>          pos_type;
        typedef wstreamoff                off_type;
};
```

2.13.4.3 Define the Code Conversion Facet

Just as in the first example, you have to define the actual code conversion facet. The steps are basically the same as before, too: define a new class template for the new code conversion type and specialize it. The code would look like this:

```
template <class internT, class externT, class stateT>
class UnicodeJISConversion
: public codecvt<internT, externT, stateT>
{
};

class UnicodeJISConversion<wchar_t, char, JISstate_t>
: public codecvt<wchar_t, char, JISstate_t>
{
protected:

 result do_in(JISstate_t& state,
              const char*  from,
              const char*  from_end,
              const char*& from_next,
              wchar_t*     to,
              wchar_t*     to_limit,
              wchar_t*&    to_next) const;

 result do_out(JISstate_t& state,
               const wchar_t*  from,
               const wchar_t*  from_end,
               const wchar_t*& from_next,
               char*           to,
               char*           to_limit,
               char*&          to_next) const;

 bool do_always_noconv() const throw()
 { return false; };

 int do_encoding() const throw();
 { return -1; }

};
```

In this case, the function `do_encoding()`has to return -1, which identifies the code conversion as state-dependent. Again, the functions `in()` and `out()` have to conform to the error indication policy explained under class ***codecvt*** in the *Class Reference.*

The distinguishing characteristic of a state-independent conversion is that the conversion state argument to `in()` and `out()` is used for communication between the file stream buffer and the code conversion facet. The file stream buffer is responsible for creating, maintaining, and deleting the conversion state. At the beginning, the file stream buffer creates a conversion state

object that represents the initial conversion state and hands it over to the code conversion facet. The facet modifies it according to the conversion it performs. The file stream buffer receives it and stores it between two subsequent code conversions.

2.13.4.4 Use the New Code Conversion Facet

Here is an example of how the new code conversion facet can be used:

```
typedef basic_fstream<wchar_t,JIS_char_traits> JIS_fstream;   \\1
JIS_fstream inout("/tmp/fil");
UnicodeJISConversion<wchar_t,char,JISstate_t> cvtfac;
locale cvtloc(locale(),&cvtfac);
inout.rdbuf()->pubimbue(cvtloc)                               \\2
wcout << inout.rdbuf();                                       \\3
```

`//1` Our Unicode-JIS code conversion needs a conversion state type different from the default type `mbstate_t`. Since the conversion state type is contained in the character traits, we have to create a new file type. Instead of `JIS_char_traits`, we could have taken advantage of the non-standard extension to the character traits template and have used `char_traits<wchar_t,JISstate_t>`.

`//2` Here the stream buffer's locale is replaced by a copy of the global locale that has a Unicode-JIS code conversion facet.

`//3` The content of the JIS encoded file `"/tmp/fil"` is read, automatically converted to Unicode, and written to `wcout`.

2.14 Differences between Standard and Traditional Iostreams[45]

The standard iostreams facility differs substantially from the traditional iostreams. This section briefly describes the main differences.

2.14.1 The Character Type

In the past, you may already have used iostreams—the traditional iostreams. The iostreams included in the Standard C++ Library are mostly compatible, yet slightly different from what you know. The most apparent change is that the new iostream classes are templates, taking the type of the character as a template parameter.

The traditional iostreams were of limited use. They could handle only byte streams; in other words, they read files byte per byte, and worked internally with a buffer of bytes. They had problems with languages that have alphabets containing thousands of characters. These alphabets are encoded

---

[45] The list in this section is not meant to be complete, since this is a preliminary release of the *User's Guide*.

as multibytes for storage on external devices like files, and represented as wide characters internally. They required a code conversion with each input and output.

The new templatized iostreams can handle large alphabets. These iostreams can be instantiated for one-byte skinny characters of type `char`, and for wide characters of type `wchar_t`. In fact, you can instantiate iostream classes for any user-defined character type. Section 2.13 describes in detail how this can be done.

2.14.2 Internationalization

Another new feature of the standard iostreams is internationalization. Traditional iostreams were incapable of adjusting to local conventions. Output of numerical items was always done following the US English conventions for number formatting. The new iostreams are internationalized to allow for local conventions. They use the standard locales described in the section on locales.

2.14.3 File Streams

2.14.3.1 Connecting Files and Streams

The traditional iostreams supported a file stream constructor, taking a file descriptor that allowed connection of a file stream to an already open file. This is no longer available in the standard iostreams.

The functions `attach()` and `detach()` do not exist anymore.

2.14.3.2 The File Buffer

Due to changes in the iostream architecture, the file buffer is now contained as a data member in the file stream classes. In some old implementations, the buffer was inherited from a base class called `fstreambase`.

The old file streams had a destructor; the new file streams don't need one. Flushing the buffer and closing the file is now done by the file buffer's destructor.

2.14.4 String Streams

Output string streams are always dynamic. The `str()` function does not have the functionality of freezing the string stream anymore. Instead, the string provided through `str()` is copied into the internal buffer; it is not used *as* the internal buffer. Accordingly, the string returned through `str()` is always a copy of the internal buffer.

If you need to influence a string stream's internal buffering, you must do it through `pubsetbuf()`.

The classes `strstream`, `istrstream`, `ostrstream`, and `strstreambuf` are deprecated features in the standard iostreams. They are still provided by this implementation of the standard iostreams, but will be omitted in the future.

2.14.5 Streams with Assign

The classes `ostream_withassign`, `istream_withassign`, and `iostream_withassign` do not exist in the standard iostreams anymore. You can only assign the data components of a stream to another stream. This is done through the functions `copyfmt()` and `rdbuf()` in class `basic_ios`.

2.15 Differences between Standard and Rogue Wave IOStreams

This section describes how the Rogue Wave implementation of the standard iostreams differs from the ISO / ANSI Standard C++ Library specification. You must be aware that whenever you use one of the features described here, the portability of your program will be impaired. It will not conform to the standard.

2.15.1 Extensions

Rogue Wave's implementation of the standard iostreams has several extensions that we will describe briefly in the sections below.

2.15.1.1 File Descriptors

The traditional iostreams allowed a file stream to connect to a file using *a file descriptor*. File descriptors are used by functions like `open()`, `close()`, `read()`, and `write()` that are part of most C libraries, especially on UNIX-based platforms. However, the ISO / ANSI standard for the programming language C and its library does not include these functions, nor does it mention file descriptors. In this sense, the use of file descriptors introduces platform and operating system dependencies into your program. This is exactly why the standard iostreams does not use file descriptors.

Now you might already have programs that use the file descriptor features of the traditional iostreams. And you may need to access system-specific files like pipes, which are accessible only through file descriptors. To address these concerns, Rogue Wave's implementation offers additional constructors and member functions in the file stream and file buffer classes that enable you to work with file descriptors.

The main additions are:

- Constructors that take a file descriptor rather than a file name;

- An additional third parameter that allows specification of file access rights. This parameter, available on several constructors and the `open()` member functions, is not available with the standard interface. The parameter has a default, so that you usually need not worry about file protection.

2.15.2 Restrictions

Rogue Wave's implementation of the standard iostreams has several restrictions, most of which correspond to the limited capabilities of current compilers in handling Standard C++. These restrictions include:

- Member templates;
- Explicit template argument specification (`use_facet` and `has_facet` in locale)

2.15.3 Deprecated Features

- The `strsteam` classes

Appendix: Implementation Dependencies and Open Issues in the Standard

Implementation-Dependent Behavior

1. The memory allocation strategy of `operator<<()` for strings is not specified; in other words, the strategy is implementation-dependent.
2. The behavior of an input file stream's `sync()` function is implementation-dependent. An input file stream's `sync()` function should synchronize the input sequence with the input file; that is, it should refill the buffer from the current position on, or the draft should state exactly what the `sync()` function does.
3. It is unspecified—that is, implementation-dependent—how `pword()` and `iword()` indicate failure. Possible choices could be throwing a `bad_alloc` exception, or setting `failbit`.

   Rogue Wave's implementation uses operator `new` for allocating these arrays, which means that `bad_alloc` will be thrown.
4. It is unspecified what happens if `iword()` or `pword()` are provided with an index that was not returned by a previous call to `xalloc()`.

   Rogue Wave's implementation allocates as much memory as necessary to provide the requested array entry.

Open Issues in the Standard

1. A call to a stream's `imbue()` function changes the stream buffer's locale object as well. This is a problem when two streams share a stream buffer and the streams' locales have different code conversion facets.
2. It is unspecified when calls to a stream's `imbue()` function are allowed. This is especially crucial when code conversion needs to be performed.