Appendix D

Locales

When in Rome,
do as the Romans do.
– proverb

Handling cultural differences — class ***locale*** — named locales — constructing locales — copying and comparing locales — the ***global***() and ***classic***() locales — comparing strings — class ***facet*** — accessing facets in a locale — a simple user-defined facet — standard facets — string comparison — numeric I/O — money I/O — date and time I/O — low-level time operations — a ***Date*** class — character classification — character code conversion — message catalogs — advice — exercises.

D.1 Handling Cultural Differences

A ***locale*** is an object that represents a set of cultural preferences, such as how strings are compared, the way numbers appear as human-readable output, and the way characters are represented in external storage. The notion of a locale is extensible so that a programmer can add new ***facet***s to a ***locale*** representing locale-specific entities not directly supported by the standard library, such as postal codes (zip codes) and phone numbers. The primary use of ***locale***s in the standard library is to control the appearance of information put to an ***ostream*** and the format accepted by an ***istream***.

Section §21.7 describes how to change ***locale*** for a stream; this appendix describes how a ***locale*** is constructed out of ***facet***s and explains the mechanisms through which a ***locale*** affects its stream. This appendix also describes how ***facet***s are defined, lists the standard ***facet***s that define specific properties of a stream, and presents techniques for implementing and using ***locale***s and ***facet***s. The standard library facilities for representing data and time are discussed as part of the presentation of date I/O.

The discussion of locales and facets is organized like this:

- §D.1 introduces the basic ideas for representing cultural differences using locales.
- §D.2 presents the ***locale*** class.
- §D.3 presents the ***facet*** class.
- §D.4 gives an overview of the standard ***facet***s and presents details of each:
  - §D.4.1 String comparison
  - §D.4.2 Input and output of numeric values
  - §D.4.3 Input and output of monetary values
  - §D.4.4 Input and output of dates and time
  - §D.4.5 Character classification
  - §D.4.6 Character code conversions
  - §D.4.7 Message catalogs

The notion of a locale is not primarily a C++ notion. Most operating systems and application environments have a notion of locale. Such a notion is – in principle – shared among all programs on a system, independently of which programming language they are written in. Thus, the C++ standard library notion of a locale can be seen as a standard and portable way for C++ programs to access information that has very different representations on different systems. Among other things, a C++ ***locale*** is a common interface to system information that is represented in incompatible ways on different systems.

D.1.1 Programming Cultural Differences

Consider writing a program that needs to be used in several countries. Writing a program in a style that allows that is often called ''internationalization'' (emphasizing the use of a program in many countries) or ''localization'' (emphasizing the adaptation of a program to local conditions). Many of the entities that a program manipulates will conventionally be displayed differently in those countries. We can handle this by writing our I/O routines to take this into account. For example:

```
void print_date(const Date& d) // print in the appropriate format
{
    switch(where_am_I) {      // user-defined style indicator
    case DK:                  // e.g., 7. marts 1999
        cout << d.day() << ". " << dk_month[d.month()] << " " << d.year();
        break;
    case UK:                  // e.g., 7 / 3 / 1999
        cout << d.day() << " / " << d.month() << " / " << d.year();
        break;
    case US:                  // e.g., 3/7/1999
        cout << d.month() << "/" << d.day() << "/" << d.year();
        break;
    // ...
    }
}
```

This style of code does the job. However, it's rather ugly, and we have to use this style consistently to ensure that all output is properly adjusted to local conventions. Worse, if we want to add a new way of writing a date, we must modify the code. We could imagine handling this problem by

creating a class hierarchy (§12.2.4). However, the information in a ***Date*** is independent of the way we want to look at it. Consequently, we don't want a hierarchy of ***Date*** types: for example, ***US_date***, ***UK_date***, and ***JP_date***. Instead, we want a variety of ways of displaying ***Date***s: for example, US-style output, UK-style output, and Japanese-style output; see §D.4.4.5.

Other problems arise with the ''let the user write I/O functions that take care of cultural differences'' approach:

[1] An application programmer cannot easily, portably, and efficiently change the appearance of built-in types without the help of the standard library.

[2] Finding every I/O operation (and every operation that prepares data for I/O in a locale-sensitive manner) in a large program is not always feasible.

[3] Sometimes, we cannot rewrite a program to take care of a new convention – and even if we could, we'd prefer a solution that didn't involve a rewrite.

[4] Having each user design and implement a solution to the problems of different cultural convention is wasteful.

[5] Different programmers will handle low-level cultural preferences in different ways, so programs dealing with the same information will differ for non-fundamental reasons. Thus, programmers maintaining code from a number of sources will have to learn a variety of programming conventions. This is tedious and error prone.

Consequently, the standard library provides an extensible way of handling cultural conventions. The iostreams library (§21.7) relies on this framework to handle both built-in and user-defined types. For example, consider a simple loop copying (***Date***, ***double***) pairs that might represent a series of measurements or a set of transactions:

```
void cpy(istream& is, ostream& os)   // copy (Date,double) stream
{
    Date d;
    double volume;

    while (is >> d >> volume) os << d << ' ' << volume << '\n';
}
```

Naturally, a real program would do something with the records, and ideally also be a bit more careful about error handling.

How would we make this program read a file that conformed to French conventions (where comma is the character used to represent the decimal point in a floating-point number; for example, *12,5* means twelve and a half) and write it according to American conventions? We can define ***locale***s and I/O operations so that ***cpy***() can be used to convert between conventions:

```
void f(istream& fin, ostream& fout, istream& fin2, ostream& fout2)
{
    fin.imbue(locale("en_US"));        // American English
    fout.imbue(locale("fr"));          // French
    cpy(fin,fout);                     // read American English, write French

    fin2.imbue(locale("fr"));          // French
    fout2.imbue(locale("en_US"));      // American English
    cpy(fin2,fout2);                   // read French, write American English
}
```

Given streams,

```
Apr 12, 1999  1000.3
Apr 13, 1999  345.45
Apr 14, 1999  9688.321
 . . .

3 juillet 1950  10,3
3 juillet 1951  134,45
3 juillet 1952  67,9
 . . .
```

this program would produce:

```
12 avril 1999 1000,3
13 avril 1999 345,45
14 avril 1999 9688,321
 . . .

July 3, 1950 10.3
July 3, 1951 134.45
July 3, 1952 67.9
 . . .
```

Much of the rest of this appendix is devoted to describing the mechanisms that make this possible and explaining how to use them. Please note that most programmers will have little reason to deal with the details of ***locale***s. Many programmers will never explicitly manipulate a ***locale***, and most who do will just retrieve a standard locale and imbue a stream with it (§21.7). However, the mechanisms provided to compose those ***locale***s and to make them trivial to use constitute a little programming language of their own.

If a program or a system is successful, it will be used by people with needs and preferences that the original designers and programmers didn't anticipate. Most successful programs will be run in countries where (natural) languages and character sets differ from those familiar to the original designers and programmers. Wide use of a program is a sign of success, so designing and programming for portability across linguistic and cultural borders is to prepare for success.

The concept of localization (internationalization) is simple. However, practical constraints make the design and implementation of ***locale*** quite intricate:

[1] A ***locale*** encapsulates cultural conventions, such as the appearance of a date. Such conventions vary in many subtle and unsystematic ways. These conventions have nothing to do with programming languages, so a programming language cannot standardize them.

[2] The concept of a ***locale*** must be extensible, because it is not possible to enumerate every cultural convention that is important to every C++ user.

[3] A ***locale*** is used in I/O operations from which people demand run-time efficiency.

[4] A ***locale*** must be invisible to the majority of programmers who want to benefit from stream I/O ''doing the right thing'' without having to know exactly what that is or how it is achieved.

[5] A ***locale*** must be available to designers of facilities that deal with cultural-sensitive information beyond the scope of the stream I/O library.

Designing a program doing I/O requires a choice between controlling formatting through ''ordinary

code'' and the use of ***locale***s. The former (traditional) approach is feasible where we can ensure that every input operation can be easily converted from one convention to another. However, if the appearance of built-in types needs to vary, if different character sets are needed, or if we need to choose among an extensible set of I/O conventions, the ***locale*** mechanism begins to look attractive.

A ***locale*** is composed of ***facet***s that control individual aspects, such as the character used for punctuation in the output of a floating-point value (***decimal_point***( ); §D.4.2) and the format used to read a monetary value (***moneypunct***; §D.4.3). A ***facet*** is an object of a class derived from class ***locale*::*facet*** (§D.3). We can think of a ***locale*** as a container of ***facet***s (§D.2, §D.3.1).

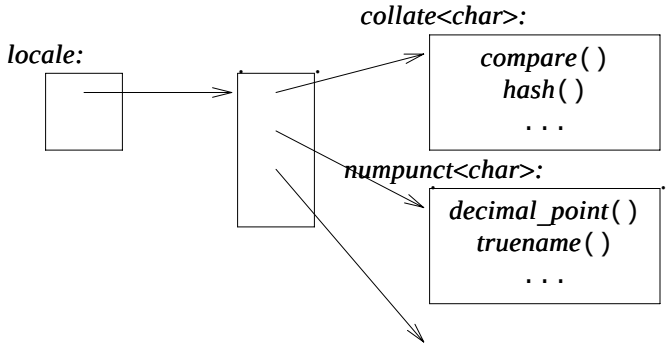
D.2 The *locale* Class

The ***locale*** class and its associated facilities are presented in <***locale***>:

```
class std::locale {
public:
    class facet;                // type used to represent aspects of a locale; §D.3
    class id;                   // type used to identify a locale; §D.3
    typedef int category;       // type used to group/categorize facets

    static const category       // the actual values are implementation defined
        none = 0,
        collate = 1,
        ctype = 1<<1,
        monetary = 1<<2,
        numeric = 1<<3,
        time = 1<<4,
        messages = 1<<5,
        all = collate | ctype | monetary | numeric | time | messages;

    locale() throw();                   // copy of global locale (§D.2.1)
    locale(const locale& x) throw();    // copy of x
    explicit locale(const char* p);     // copy of locale named p (§D.2.1)

    ~locale() throw();

    locale(const locale& x, const char* p, category c);         // copy of x plus facets from p's c
    locale(const locale& x, const locale& y, category c);       // copy of x plus facets from y's c

    template <class Facet> locale(const locale& x, Facet* f);   // copy of x plus facet f
    template <class Facet> locale combine(const locale& x);     // copy of *this plus Facet from x

    const locale& operator=(const locale& x) throw();

    bool operator==(const locale&) const;       // compare locales
    bool operator!=(const locale&) const;

    string name() const;                        // name of this locale (§D.2.1)

    template <class Ch, class Tr, class A>      // compare strings using this locale
    bool operator()(const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&) const;
```

```
    static locale global(const locale&);      // set global locale and return old global locale
    static const locale& classic();           // get ''classic'' C-style locale
private:
    // representation
};
```

A *locale* can be thought of as an interface to a *map<id,facet*>*; that is, something that allows us to use a *locale::id* to find a corresponding object of a class derived from *locale::facet*. A real implementation of *locale* is an efficient variant of this idea. The layout will be something like this:



Here, *collate<char>* and *numpunct<char>* are standard library facets (§D.4). As all facets, they are derived from *locale::facet*.

A *locale* is meant to be copied freely and cheaply. Consequently, a *locale* is almost certainly implemented as a handle to the specialized *map<id,facet*>* that constitutes the main part of its implementation. The *facet*s must be quickly accessible in a *locale*. Consequently, the specialized *map<id,facet*>* will be optimized to provide array-like fast access. The *facet*s of a *locale* are accessed by using the *use_facet<Facet>(loc)* notation; see §D.3.1.

The standard library provides a rich set of *facet*s. To help the programmer manipulate *facet*s in logical groups, the standard *facet*s are grouped into categories, such as *numeric* and *collate* (§D.4).

A programmer can replace *facet*s from existing categories (§D.4, §D.4.2.1). However, it is not possible to add new categories; there is no way for a programmer to define a new category. The notion of ''category'' applies to standard library facets only, and it is not extensible. Thus, a facet need not belong to any category, and many user-defined facets do not.

By far the dominant use of *locale*s is implicitly, in stream I/O. Each *istream* and *ostream* has its own *locale*. The *locale* of a stream is by default the global *locale* (§D.2.1) at the time of the stream's creation. The *locale* of a stream can be set by the *imbue()* operation and we can extract a copy of a stream's *locale* using *getloc()* (§21.6.3).

D.2.1 Named Locales

A *locale* is constructed from another *locale* and from *facet*s. The simplest way of making a locale is to copy an existing one. For example:

```
locale loc0;                        // copy of the current global locale (§D.2.3)

locale loc1 = locale();             // copy of the current global locale (§D.2.3)
locale loc2("");                    // copy of ''the user's preferred locale''

locale loc3("C");                   // copy of the "C" locale
locale loc4 = locale::classic();    // copy of the "C" locale

locale loc5("POSIX");               // copy of the implementation-defined "POSIX" locale
```

The meaning of ***locale*("*C*")** is defined by the standard to be the ''classic'' C locale; this is the locale that has been used throughout this book. Other ***locale*** names are implementation defined.

The ***locale*("")** is deemed to be ''the user's preferred locale.'' This locale is set by extralinguistic means in a program's execution environment.

Most operating systems have ways of setting a locale for a program. Often, a locale suitable to the person using a system is chosen when that person first encounters a system. For example, I would expect a person who configures a system to use Argentine Spanish as its default setting will find ***locale*("")** to mean ***locale*("*es_AR*")**. A quick check on one of my systems revealed 51 locales with mnemonic names, such as ***POSIX***, ***de***, ***en_UK***, ***en_US***, ***es***, ***es_AR***, ***fr***, ***sv***, ***da***, ***pl***, and ***iso_8859_1***. POSIX recommends a format of a lowercase language name, optionally followed by an uppercase country name, optionally followed by an encoding specifier; for example, ***jp_JP.jit***. However, these names are not standardized across platforms. On another system, among many other locale names, I found ***g***, ***uk***, ***us***, ***s***, ***fr***, ***sw***, and ***da***. The C++ standard does not define the meaning of a ***locale*** for a given country or language, though there may be platform-specific standards. Consequently, to use named ***locale***s on a given system, a programmer must refer to system documentation and experiment.

It is generally a good idea to avoid embedding ***locale*** name strings in the program text. Mentioning a file name or a system constant in the program text limits the portability of a program and often forces a programmer who wants to adapt a program to a new environment to find and change such values. Mentioning a locale name string has similar unpleasant consequences. Instead, locales can be picked up from the program's execution environment (for example, using ***locale*("")**), or the program can request an expert user to specify alternative locales by entering a string. For example:

```
void user_set_locale(const string& question_string)
{
    cout << question_string;    // e.g., "If you want to use a different locale, please enter its name"
    string s;
    cin >> s;
    locale::global(locale(s.c_str()));  // set global locale as specified by user
}
```

It is usually better to let a non-expert user pick from a list of alternatives. A routine for doing this would need to know where and how a system kept its locales.

If the string argument doesn't refer to a defined ***locale***, the constructor throws the ***runtime_error*** exception (§14.10). For example:

```
void set_loc(locale& loc, const char* name)
try
{
    loc = locale(name);
}
catch (runtime_error) {
    cerr << "locale \"" << name << "\" isn't defined\n";
    // ...
}
```

If a ***locale*** has a name string, ***name***() will return it. If not, ***name***() will return ***string***("*"). A name string is primarily a way to refer to a ***locale*** stored in the execution environment. Secondarily, a name string can be used as a debugging aid. For example:

```
void print_locale_names(const locale& my_loc)
{
    cout << "name of current global locale: " << locale().name() << "\n";
    cout << "name of classic C locale: " << locale::classic().name() << "\n";
    cout << "name of ''user's preferred locale'': " << locale("").name() << "\n";
    cout << "name of my locale: " << my_loc.name() << "\n";
}
```

Locales with identical name strings different from the default ***string***("*") compare equal. However, == or != provide more direct ways of comparing locales.

The copy of a ***locale*** with a name string gets the same name as that ***locale*** (if it has one), so many ***locale***s can have the same name string. That's logical because ***locale***s are immutable, so all of these objects define the same set of cultural conventions.

A call ***locale***(*loc*,"*Foo*",*cat*) makes a locale that is like ***loc*** except that it takes the facets from the category ***cat*** of ***locale***("*Foo*"). The resulting locale has a name string if and only if ***loc*** has one. The standard doesn't specify exactly which name string the new locale gets, but it is supposed to be different from ***loc***'s. One obvious implementation would be to compose the new string out of ***loc***'s name string and "*Foo*". For example, if ***loc***'s name string is ***en_UK***, the new locale may have "*en_UK*:*Foo*" as its name string.

The name strings for a newly created ***locale*** can be summarized like this:

Locale	Name String
locale("Foo")	"Foo"
locale(loc)	*loc.name*()
locale(loc,"Foo",cat)	New name string if ***loc*** has a name string; otherwise, ***string***("*")
locale(loc,loc2,cat)	New name string if ***loc*** and *loc2* have strings; otherwise, ***string***("*")
locale(loc,Facet)	***string***("*")
loc.combine(loc2)	***string***("*")

There are no facilities for a programmer to specify a C-style string as a name for a newly created ***locale*** in a program. Name strings are either defined in the program's execution environment or created as combinations of such names by ***locale*** constructors.

D.2.1.1 Constructing New Locales

A new locale is made by taking an existing ***locale*** and adding or replacing ***facet***s. Typically, a new ***locale*** is a minor variation on an existing one. For example:

```
void f(const locale& loc, const My_money_io* mio) // My_money_io defined in §D.4.3.1
{
    locale loc1(locale("POSIX"),loc,locale::monetary);    // use monetary facets from loc
    locale loc2 = locale(locale::classic(), mio);         // classic plus mio
    // ...
}
```

Here, ***loc1*** is a copy of the ***POSIX*** locale modified to use ***loc***'s monetary facets (§D.4.3). Similarly, ***loc2*** is a copy of the ***C*** locale modified to use a ***My_money_io*** (§D.4.3.1). If a ***Facet**** argument (here, ***My_money_io***) is ***0***, the resulting ***locale*** is simply a copy of the ***locale*** argument.

When using

```
locale(const locale& x, Facet* f);
```

the ***f*** argument must identify a specific facet type. A plain ***facet**** is not sufficient. For example:

```
void g(const locale::facet* mio1, const My_money_io* mio2)
{
    locale loc3 = locale(locale::classic(), mio1); // error: type of facet not known
    locale loc4 = locale(locale::classic(), mio2); // ok: type of facet known
    // ...
}
```

The reason is that the ***locale*** uses the type of the ***Facet**** argument to determine the type of the facet at compile time. Specifically, the implementation of ***locale*** uses a facet's identifying type, ***facet::id*** (§D.3), to find that facet in the locale (§D.3.1).

Note that the

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

constructor is the only mechanism offered within the language for the programmer to supply a ***facet*** to be used through a ***locale***. Other ***locale***s are supplied by implementers as named locales (§D.2.1). These named locales can be retrieved from the program's execution environment. A programmer who understands the implementation-specific mechanism used for that might be able to add new ***locale***s that way (§D.6[11,12]).

The set of constructors for ***locale*** is designed so that the type of every ***facet*** is known either from type deduction (of the ***Facet*** template parameter) or because it came from another ***locale*** (that knew its type). Specifying a ***category*** argument specifies the type of ***facet***s indirectly, because the ***locale*** knows the type of the ***facet***s in the categories. This implies that the ***locale*** class can (and does) keep track of the types of ***facet*** types so that it can manipulate them with minimal overhead.

The ***locale::id*** member type is used by ***locale*** to identify ***facet*** types (§D.3).

It is sometimes useful to construct a ***locale*** that is a copy of another except for a ***facet*** copied from yet another ***locale***. The ***combine()*** template member function does that. For example:

```
void f(const locale& loc, const locale& loc2)
{
    locale loc3 = loc.combine< My_money_io >(loc2);
    // ...
}
```

The resulting ***loc3*** behaves like ***loc*** except that it uses a copy of ***My_money_io*** (§D.4.3.1) from ***loc2*** to format monetary I/O. If ***loc2*** doesn't have a ***My_money_io*** to give to the new ***locale***, ***combine***( ) will throw a ***runtime_error*** (§14.10). The result of ***combine***( ) has no name string.

D.2.2 Copying and Comparing Locales

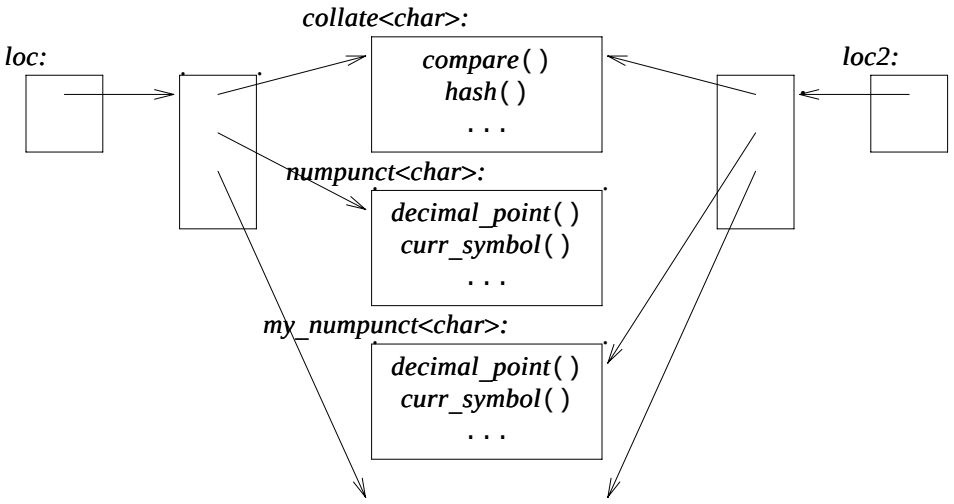
A ***locale*** can be copied by initialization and by assignment. For example:

```
void swap(locale& x, locale& y)        // just like std::swap()
{
    locale temp = x;
    x = y;
    y = temp;
}
```

The copy of a ***locale*** compares equal to the original, but the copy is an independent and separate object. For example:

```
void f(locale* my_locale)
{
    locale loc = locale::classic();  // "C" locale

    if (loc != locale::classic()) {
        cerr << "implementation error: send bug report to vendor\n";
        exit(1);
    }

    if (&loc != &locale::classic()) cout << "no surprise: addresses differ\n";

    locale loc2 = locale(loc, my_locale, locale::numeric);

    if (loc == loc2) {
        cout << "my numeric facets are the same as classic()'s numeric facets\n";
        // ...
    }

    // ...
}
```

If ***my_locale*** has a numeric punctuation facet, ***my_numpunct<char>***, that is different from ***classic***( )'s standard ***numpunct<char>***, the resulting ***locale***s can be represented like this:



There is no way of modifying a ***locale***. Instead, the ***locale*** operations provide ways of making new ***locales*** from existing ones. The fact that a ***locale*** is immutable after it has been created is essential for run-time efficiency. This allows someone using a ***locale*** to call virtual functions of a ***facet*** and to cache the values returned. For example, an ***istream*** can know what character is used to represent the decimal point and how ***true*** is represented, without calling ***decimal_point***( ) each time it reads a number and ***truename***( ) each time it reads to a ***bool*** (§D.4.2). Only a call of ***imbue***( ) for the stream (§21.6.3) can cause such calls to return a different value.

D.2.3 The *global()* and the *classic()* Locales

The notion of a current locale for a program is provided by ***locale***( ), which yields a copy of the current locale, and ***locale***::***global***(***x***), which sets the current locale to ***x***. The current locale is commonly referred to as the ''global locale,'' reflecting its probable implementation as a global (or ***static***) object.

The global locale is implicitly used when a stream is initialized. That is, every new stream is imbued (§21.1, §21.6.3) with a copy of ***locale***( ). Initially, the global locale is the standard C locale, ***locale***::***classic***( ).

The ***locale***::***global***( ) static member function allows a programmer to specify a locale to be used as the global locale. A copy of the previous global locale is returned by ***global***( ). This allows a user to restore the global locale. For example:

```
void f(const locale& my_loc)
{
    ifstream fin1(some_name);                    // fin1 is imbued with the global locale
    locale& old_global = locale::global(my_loc); // set new global locale
    ifstream fin2(some_other_name);              // fin2 is imbued with my_loc
    // ...
    locale::global(old_global);                  // restore old global locale
}
```

If a locale *x* has a name string, ***locale*::*global*(*x*)** also sets the C global locale. This implies that if a C++ program calls a locale-sensitive function from the C standard library, the treatment of locale will be consistent throughout a mixed C and C++ program.

If a locale *x* does not have a name string, it is undefined whether ***locale*::*global*(*x*)** affects the C global locale. This implies that a C++ program cannot reliably and portably set the C locale to a locale that wasn't retrieved from the execution environment. There is no standard way for a C program to set the C++ global locale (except by calling a C++ function to do so). In a mixed C and C++ program, having the C global locale differ from ***global*()** is error prone.

Setting the global locale does not affect existing I/O streams; those still use the locales that they were imbued with before the global locale was reset. For example, ***fin1*** is unaffected by the manipulation of the global locale that caused ***fin2*** to be imbued with ***my_loc***.

Changing the global locale suffers the same problems as all other techniques relying on changing global data: It is essentially impossible to know what is affected by a change. It is therefore best to reduce use of ***global*()** to a minimum and to localize those changes in a few sections of code that obey a simple strategy for the changes. The ability to imbue (§21.6.3) individual streams with specific ***locale***s makes that easier. However, too many explicit uses of ***locale***s and ***facet***s scattered throughout a program will also become a maintenance problem.

D.2.4 Comparing Strings

Comparing two strings according to a ***locale*** is possibly the most common *explicit* use of a ***locale***. Consequently, this operation is provided directly by ***locale*** so that users don't have to build their own comparison function from the ***collate*** facet (§D.4.1). To be directly useful as a predicate (§18.4.2), this comparison function is defined as ***locale***'s ***operator*()()**. For example:

```
void f(vector<string>& v, const locale& my_locale)
{
    sort(v.begin(),v.end());                // sort using < to compare elements
    // ...
    sort(v.begin(),v.end(),my_locale);      // sort according to the rules of my_locale
    // ...
}
```

By default, the standard library ***sort*()** uses < for the numerical value of the implementation character set to determine collation order (§18.7, §18.6.3.1).

Note that ***locale***s compare ***basic_string***s rather than C-style strings.

D.3 Facets

A ***facet*** is an object of a class derived from ***locale***'s member class ***facet***:

```
class std::locale::facet {
protected:
    explicit facet(size_t r = 0);   // "r==0": the locale controls the lifetime of this facet
    virtual ~facet();               // note: protected destructor
```

```
private:
    facet(const facet&);              // not defined
    void operator=(const facet&);     // not defined

    // representation
};
```

The copy operations are ***private*** and are left undefined to prevent copying (§11.2.2).

The ***facet*** class is designed to be a base class and has no public functions. Its constructor is ***protected*** to prevent the creation of ''plain ***facet***'' objects, and its destructor is virtual to ensure proper destruction of derived-class objects.

A ***facet*** is intended to be managed through pointers by ***locale***s. A ***0*** argument to the ***facet*** constructor means that ***locale*** should delete the ***facet*** when the last reference to it goes away. Conversely, a nonzero constructor argument ensures that ***locale*** never deletes the ***facet***. A nonzero argument is meant for the rare case in which the lifetime of a facet is controlled directly by the programmer rather than indirectly through a locale. For example, we could try to create objects of the standard facet type ***collate_byname<char>*** (§D.4.1.1) like this:

```
void f(const string& s1, const string& s2)
{
    // normal case: (default) argument 0 means that locale is responsible for deletion:
    collate<char>* p = new collate_byname<char>("pl");
    locale loc(locale(),p);

    // rare case: argument 1 means that user is responsible for deletion:
    collate<char>* q = new collate_byname<char>("ge",1);

    collate_byname<char> bug1("sw");        // error: cannot destroy local variable
    collate_byname<char> bug2("no",1);      // error: cannot destroy local variable

    // ...

    // q cannot be deleted: collate_byname<char>'s destructor is protected
    // no delete p; locale manages deletion of *p
}
```

That is, standard facets are useful when managed by locales, as base classes, and only rarely in other ways.

A ***_byname()*** facet is a facet from a named locale in the the execution environment (§D.2.1).

For a ***facet*** to be found in a ***locale*** by ***has_facet()*** and ***use_facet()*** (§D.3.1), each kind of facet must have an ***id***:

```
class std::locale::id {
public:
    id();
private:
    id(const id&);                  // not defined
    void operator=(const id&);      // not defined

    // representation
};
```

The copy operations are declared private and are left undefined to prevent copying (§11.2.2).

The intended use of ***id*** is for the user to define a ***static*** member of type ***id*** of each class supplying a new ***facet*** interface (for example, see §D.4.1). The ***locale*** mechanisms use ***id***s to identify facets (§D.2, §D.3.1). In the obvious implementation of a ***locale***, an ***id*** is used as an index into a vector of pointers to facets, thereby implementing an efficient ***map<id,facet*>***.

Data used to define a (derived) ***facet*** is defined in the derived class rather than in the base class ***facet*** itself. This implies that the programmer defining a ***facet*** has full control over the data and that arbitrary amounts of data can be used to implement the concept represented by a ***facet***.

Note that all member functions of a user-defined ***facet*** should be ***const*** members. Generally, a facet is intended to be immutable (§D.2.2).

D.3.1 Accessing Facets in a Locale

The facets of a ***locale*** are accessed through the template function ***use_facet***, and we can inquire whether a ***locale*** has a specific ***facet***, using the template function ***has_facet***:

```
template <class Facet> bool has_facet(const locale&) throw();
template <class Facet> const Facet& use_facet(const locale&);  // may throw bad_cast
```

Think of these template functions as doing a lookup in their ***locale*** argument for their template parameter ***Facet***. Alternatively, think of ***use_facet*** as a kind of explicit type conversion (cast) of a ***locale*** to a specific ***facet***. This is feasible because a ***locale*** can have only one ***facet*** of a given type. For example:

```
void f(const locale& my_locale)
{
    char c = use_facet< numpunct<char> >(my_locale).decimal_point() // use standard facet
    // ...

    if (has_facet<Encrypt>(my_locale)) {      // does my_locale contain an Encrypt facet?
        const Encrypt& f = use_facet<Encrypt>(my_locale);   // retrieve Encrypt facet
        const Crypto c = f.get_crypto();                    // use Encrypt facet
        // ...
    }
    // ...
}
```

Note that ***use_facet*** returns a reference to a ***const*** facet, so we cannot assign the result of ***use_facet*** to a non-***const***. This makes sense because a facet is meant to be immutable and to have only ***const*** members.

If we try ***use_facet<X>(loc)*** and ***loc*** doesn't have an ***X*** facet, ***use_facet()*** throws ***bad_cast*** (§14.10). The standard ***facet***s are guaranteed to be available for all locales (§D.4), so we don't need to use ***has_facet*** for standard facets. For standard facets, ***use_facet*** will not throw ***bad_cast***.

How might ***use_facet*** and ***has_facet*** be implemented? Remember that we can think of a ***locale*** as a ***map<id,facet*>*** (§D.2). Given a ***facet*** type as the ***Facet*** template argument, the implementation of ***has_facet*** or ***use_facet*** can refer to ***Facet::id*** and use that to find the corresponding facet. A very naive implementation of ***has_facet*** and ***use_facet*** might look like this:

```
// pseudoimplementation: imagine that a locale has a map<id,facet*> called facet_map

template <class Facet> bool has_facet(const locale& loc) throw()
{
    const locale::facet* f = loc.facet_map[Facet::id];
    return f ? true : false;
}

template <class Facet> const Facet& use_facet(const locale& loc)
{
    const locale::facet* f = loc.facet_map[Facet::id];
    if (f) return static_cast<const Facet&>(*f);
    throw bad_cast();
}
```

Another way of looking at the ***facet*::*id*** mechanism is as an implementation of a form of compile-time polymorphism. A ***dynamic_cast*** can be used to get very similar results to what ***use_facet*** produces. However, the specialized ***use_facet*** can be implemented more efficiently than the more general ***dynamic_cast***.

An ***id*** really identifies an interface and a behavior rather than a class. That is, if two facet classes have exactly the same interface and implement the same semantics (as far as a ***locale*** is concerned), they should be identified by the same ***id***. For example, ***collate<char>*** and ***collate_byname<char>*** are interchangeable in a ***locale***, so both are identified by ***collate<char>*::*id*** (§D.4.1).

If we define a ***facet*** with a new interface – such as ***Encrypt*** in ***f*()** – we must define a corresponding ***id*** to identify it (see §D.3.2 and §D.4.1).

D.3.2 A Simple User-Defined Facet

The standard library provides standard facets for the most critical areas of cultural differences, such as character sets and I/O of numbers. To examine the facet mechanism in isolation from the complexities of widely used types and the efficiency concerns that accompany them, let me first present a ***facet*** for a trivial user-defined type:

```
enum Season { spring, summer, fall, winter };
```

This was just about the simplest user-defined type I could think of. The style of I/O outlined here can be used with little variation for most simple user-defined types.

```
class Season_io : public locale::facet {
public:
    Season_io(int i = 0) : locale::facet(i) { }

    ~Season_io() { }      // to make it possible to destroy Season_io objects (§D.3)

    virtual const string& to_str(Season x) const = 0;          // string representation of x

    // place Season corresponding to s in x:
    virtual bool from_str(const string& s, Season& x) const = 0;
```

```
    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)
};

locale::id Season_io::id; // define the identifier object
```

For simplicity, this ***facet*** is limited to representations using ***char***.

The ***Season_io*** class provides a general and abstract interface for all ***Season_io*** facets. To define the I/O representation of a ***Season*** for a particular locale, we derive a class from ***Season_io***, defining ***to_str()*** and ***from_str()*** appropriately.

Output of a ***Season*** is easy. If the stream has a ***Season_io*** facet, we can use that to convert the value into a string. If not, we can output the ***int*** value of the ***Season***:

```
ostream& operator<<(ostream& s, Season x)
{
    const locale& loc = s.getloc(); // extract the stream's locale (§21.7.1)
    if (has_facet<Season_io>(loc)) return s << use_facet<Season_io>(loc).to_str(x);
    return s << int(x);
}
```

Note that this `<<` operator is implemented by invoking `<<` for other types. This way, we benefit from the simplicity of using `<<` compared to accessing the ***ostream***'s stream buffers directly, from the locale sensitivity of those `<<` operators, and from the error handling provided for those `<<` operators. Standard ***facet***s tend to operate directly on stream buffers (§D.4.2.2, §D.4.2.3) for maximum efficiency and flexibility, but for many simple user-defined types, there is no need to drop to the ***streambuf*** level of abstraction.

As is typical, input is a bit more complicated than output:

```
istream& operator>>(istream& s, Season& x)
{
    const locale& loc = s.getloc();        // extract the stream's locale (§21.7.1)

    if (has_facet<Season_io>(loc)) {       // read alphabetic representation
        const Season_io& f = use_facet<Season_io>(loc);
        string buf;
        if (!(s>>buf && f.from_str(buf,x))) s.setstate(ios_base::failbit);
        return s;
    }

    int i;                 // read numeric representation
    s >> i;
    x = Season(i);
    return s;
}
```

The error handling is simple and follows the error-handling style for built-in types. That is, if the input string didn't represent a ***Season*** in the chosen locale, the stream is put into the ***fail*** state. If exceptions are enabled, this implies that an ***ios_base::failure*** exception is thrown (§21.3.6).

Here is a trivial test program:

```
int main()      // trivial test
{
    Season x;

    // Use default locale (no Season_io facet) implies integer I/O:
    cin >> x;
    cout << x << endl;

    locale loc(locale(),new US_season_io);
    cout.imbue(loc);      // Use locale with Season_io facet
    cin.imbue(loc);       // Use locale with Season_io facet

    cin >> x;
    cout << x << endl;
}
```

Given the input

```
2
summer
```

this program responded:

```
2
summer
```

To get this, we must define *US_season_io* to define the string representation of the seasons and override the *Season_io* functions that convert between string representations and the enumerators:

```
class US_season_io : public Season_io {
    static const string seasons[];
public:
    const string& to_str(Season) const;
    bool from_str(const string&, Season&) const;

    // note: no US_season_io::id
};

const string US_season_io::seasons[] = { "spring", "summer", "fall", "winter" };

const string& US_season_io::to_str(Season x) const
{
    if (x<spring || winter<x) {
        static const string ss = "no-such-season";
        return ss;
    }
    return seasons[x];
}
```

```
bool US_season_io::from_str(const string& s, Season& x) const
{
    const string* beg = &seasons[spring];
    const string* end = &seasons[winter]+1;
    const string* p = find(beg,end,s);    // §3.8.1, §18.5.2
    if (p==end) return false;
    x = Season(p-beg);
    return true;
}
```

Note that because ***US_season_io*** is simply an implementation of the ***Season_io*** interface, I did not define an ***id*** for ***US_season_io***. In fact, if we want ***US_season_io*** to be used as a ***Season_io***, we must not give ***US_season_io*** its own ***id***. Operations on ***locale***s, such as ***has_facet*** (§D.3.1), rely on facets implementing the same concepts being identified by the same ***id*** (§D.3).

The only interesting implementation question was what to do if asked to output an invalid ***Season***. Naturally, that shouldn't happen. However, it is not uncommon to find an invalid value for a simple user-defined type, so it is realistic to take that possibility into account. I could have thrown an exception, but when dealing with simple output intended for humans to read, it is often helpful to produce an ''out of range'' representation for an out-of-range value. Note that for input, the error-handling policy is left to the >> operator, whereas for output, the facet function ***to_str***() implements an error-handling policy. This was done to illustrate the design alternatives. In a ''production design,'' the ***facet*** functions would either implement error handling for both input and output or just report errors for >> and << to handle.

This ***Season_io*** design relied on derived classes to supply the locale-specific strings. An alternative design would have ***Season_io*** itself retrieve those strings from a locale-specific repository (see §D.4.7). The possibility of having a single ***Season_io*** class to which the season strings are passed as constructor arguments is left as an exercise (§D.6[2]).

D.3.3 Uses of Locales and Facets

The primary use of ***locale***s within the standard library is in I/O streams. However, the ***locale*** mechanism is a general and extensible mechanism for representing culture-sensitive information. The ***messages*** facet (§D.4.7) is an example of a facet that has nothing to do with I/O streams. Extensions to the iostreams library and even I/O facilities that are not based on streams might take advantage of locales. Also, a user may use ***locale***s as a convenient way of organizing arbitrary culture-sensitive information.

Because of the generality of the ***locale***/***facet*** mechanism, the possibilities for user-defined ***facet***s are unlimited. Plausible candidates for representation as ***facet***s are dates, time zones, phone numbers, social security numbers (personal identification numbers), product codes, temperatures, general (unit,value) pairs, postal codes (zip codes), clothe sizes, and ISBN numbers.

As with every other powerful mechanism, ***facet***s should be used with care. That something can be represented as a ***facet*** doesn't mean that it is best represented that way. The key issues to consider when selecting a representation for cultural dependencies are – as ever – how the various decisions affect the difficulty of writing code, the ease of reading the resulting code, the maintainability of the resulting program, and the efficiency in time and space of the resulting I/O operations.

D.4 Standard Facets

In <*locale*>, the standard library provides these *facet*s for the *classic*( ) locale:

Standard Facets (in the *classic()* locale)			
	Category	**Purpose**	**Facets**
§D.4.1	*collate*	String comparison	*collate<Ch>*
§D.4.2	*numeric*	Numeric I/O	*numpunct<Ch>* *num_get<Ch>* *num_put<Ch>*
§D.4.3	*monetary*	Money I/O	*moneypunct<Ch>* *moneypunct<Ch,true>* *money_get<Ch>* *money_put<Ch>*
§D.4.4	*time*	Time I/O	*time_get<Ch>* *time_put<Ch>*
§D.4.5	*ctype*	Character classification	*ctype<Ch>* *codecvt<Ch,char,mbstate_t>*
§D.4.7	*messages*	Message retrieval	*messages<Ch>*

In this table, ***Ch*** is as shorthand for ***char*** or ***wchar_t***. A user who needs standard I/O to deal with another character type ***X*** must provide suitable versions of facets for ***X***. For example, ***codecvt<X, char, mbstate_t>*** (§D.4.6) might be needed to control conversions between ***X*** and ***char***. The ***mbstate_t*** type is used to represent the shift states of a multibyte character representation (§D.4.6); ***mbstate_t*** is defined in <*cwchar*> and <*wchar.h*>. The equivalent to ***mbstate_t*** for an arbitrary character type ***X*** is ***char_traits<X>::state_type***.

In addition, the standard library provides these *facet*s in <*locale*>:

Standard Facets			
	Category	**Purpose**	**Facets**
§D.4.1	*collate*	String comparison	*collate_byname<Ch>*
§D.4.2	*numeric*	Numeric I/O	*numpunct_byname<Ch>* *num_get<C,In>* *num_put<C,Out>*
§D.4.3	*monetary*	Money I/O	*moneypunct_byname<Ch,International>* *money_get<C,In>* *money_put<C,Out>*
§D.4.4	*time*	Time I/O	*time_put_byname<Ch,Out>*
§D.4.5	*ctype*	Character classification	*ctype_byname<Ch>*
§D.4.7	*messages*	Message retrieval	*messages_byname<Ch>*

When instantiating a facet from this table, ***Ch*** can be ***char*** or ***wchar_t***; ***C*** can be any character type (§20.1). ***International*** can be ***true*** or ***false***; ***true*** means that a four-character ''international''

representation of a currency symbol is used (§D.4.3.1). The *mbstate_t* type is used to represent the shift states of a multibyte character representation (§D.4.6); *mbstate_t* is defined in <*cwchar*> and <*wchar.h*>.

In and *Out* are input iterators and output iterators, respectively (§19.1, §19.2.1). Providing the *_put* and *_get* facets with these template arguments allows a programmer to provide facets that access nonstandard buffers (§D.4.2.2). Buffers associated with iostreams are stream buffers, so the iterators provided for those are *ostreambuf_iterator*s (§19.2.6.1, §D.4.2.2). Consequently, the function *failed*() (§19.2.6.1) is available for error handling.

An *F_byname* facet is derived from the facet *F*. *F_byname* provides the identical interface to *F*, except that it adds a constructor taking a string argument naming a *locale* (see §D.4.1). The *F_byname*(*name*) provides the appropriate semantics for *F* defined in *locale*(*name*). The idea is to pick a version of a standard facet from a named *locale* (§D.2.1) in the program's execution environment. For example:

```
void f(vector<string>& v, const locale& loc)
{
    locale d1(loc,new collate_byname<char>("da"));   // use Danish string comparison
    locale dk(d1,new ctype_byname<char>("da"));      // use Danish character classification
    sort(v.begin(),v.end(),dk);
    // ...
}
```

This new *dk* locale will use Danish-style strings but will retain the default conventions for numbers. Note that because *facet*'s second argument is by default *0*, the *locale* manages the lifetime of a *facet* created using operator *new* (§D.3).

Like the *locale* constructors that take string arguments, the *_byname* constructors access the program's execution environment. This implies that they are very slow compared to constructors that do not need to consult the environment. It is almost always faster to construct a locale and then to access its facets than it is to use *_byname* facets in many places in a program. Thus, reading a facet from the environment once and then using the copy in main memory repeatedly is usually a good idea. For example:

```
locale dk("da");    // read the Danish locale (incl. all of its facets) once
                    // then use the dk locale and its facets as needed

void f(vector<string>& v, const locale& loc)
{
    const collate<char>& col = use_facet< collate<char> >(dk);
    const collate<char>& ctyp = use_facet< ctype<char> >(dk);

    locale d1(loc,col);  // use Danish string comparison
    locale d2(d1,ctyp);  // use Danish character classification and Danish string comparison

    sort(v.begin(),v.end(),d2);
    // ...
}
```

The notion of categories gives a simpler way of manipulating standard facets in locales. For example, given the *dk* locale, we can construct a *locale* that reads and compares strings according to the

rules of Danish (that give three extra vowels compared to English) but that retains the syntax of numbers used in C++:

```
locale dk_us(locale::classic(),dk,collate|ctype);    // Danish letters, American numbers
```

The presentations of individual standard ***facet***s contains more examples of ***facet*** use. In particular, the discussion of ***collate*** (§D.4.1) brings out many of the common structural aspects of ***facet***s.

Note that the standard ***facet***s often depend on each other. For example, ***num_put*** depends on ***numpunct***. Only if you have a detailed knowledge of individual ***facet***s can you successfully mix and match facets or add new versions of the standard facets. In other words, beyond the simple operations mentioned in §21.7, the ***locale*** mechanisms are not meant to be directly used by novices.

The design of an individual facet is often messy. The reason is partially that facets have to reflect messy cultural conventions outside the control of the library designer, and partially that the C++ standard library facilities have to remain largely compatible with what is offered by the C standard library and various platform-specific standards. For example, POSIX provides locale facilities that it would be unwise for a library designer to ignore.

On the other hand, the framework provided by locales and facets is very general and flexible. A facet can be designed to hold any data, and the facet's operations can provide any desired operation based on that data. If the behavior of a new facet isn't overconstrained by convention, its design can be simple and clean (§D.3.2).

D.4.1 String Comparison

The standard ***collate*** facet provides ways of comparing arrays of characters of type ***Ch***:

```
template <class Ch>
class std::collate : public locale::facet {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit collate(size_t r = 0);

    int compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const
        { return do_compare(b,e,b2,e2); }

    long hash(const Ch* b, const Ch* e) const { return do_hash(b,e); }
    string_type transform(const Ch* b, const Ch* e) const { return do_transform(b,e); }

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

protected:
    ~collate();      // note: protected destructor

    virtual int do_compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const;
    virtual string_type do_transform(const Ch* b, const Ch* e) const;
    virtual long do_hash(const Ch* b, const Ch* e) const;
};
```

Like all facets, ***collate*** is publically derived from ***facet*** and provides a constructor that takes an argument that tells whether class ***locale*** controls the lifetime of the facet (§D.3).

Note that the destructor is protected. The *collate* facet isn't meant to be used directly. Rather, it is intended as a base class for all (derived) collation classes and for *locale* to manage (§D.3). Application programmers, implementation providers, and library vendors will write the string comparison facets to be used through the interface provided by *collate*.

The *compare*() function does the basic string comparison according to the rules defined for a particular *collate*; it returns *1* if the first string is lexicographically greater than the second, *0* if the strings are identical, and *-1* if the second string is greater than the first. For example:

```
void f(const string& s1, const string& s2, collate<char>& cmp)
{
    const char* cs1 = s1.data();    // because compare() operates on char[]s
    const char* cs2 = s2.data();
    switch (cmp.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size())) {
    case 0:         // identical strings according to cmp
        // ...
        break;
    case -1:        // s1 < s2
        // ...
        break;
    case 1:         // s1 > s2
        // ...
        break;
    }
}
```

Note that the *collate* member functions compare arrays of *Ch* rather than *basic_string*s or zero-terminated C-style strings. In particular, a *Ch* with the numeric value *0* is treated as an ordinary character rather than as a terminator. Also, *compare*() differs from *strcmp*(), returning exactly the values *-1*, *0*, and *1* rather than simply *0* and (arbitrary) negative and positive values (§20.4.1).

The standard library *string* is not *locale* sensitive. That is, it compares strings according to the rules of the implementation's character set (§C.2). Furthermore, the standard *string* does not provide a direct way of specifying comparison criteria (Chapter 20). To do a *locale*-sensitive comparison, we can use a *collate*'s *compare*(). Notationally, it can be more convenient to use *collate*'s *compare*() indirectly through a *locale*'s *operator*() (§D.2.4). For example:

```
void f(const string& s1, const string& s2, const char* n)
{
    bool b = s1 == s2;              // compare using implementation's character set values

    const char* cs1 = s1.data();    // because compare() operates on char[]s
    const char* cs2 = s2.data();

    typedef collate<char> Col;

    const Col& glob = use_facet<Col>(locale());         // from the current global locale
    int i0 = glob.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size());

    const Col& my_coll = use_facet<Col>(locale(""));    // from my preferred locale
    int i1 = my_coll.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size());
```

```
    const Col& coll = use_facet<Col>(locale(n));          // from locale named n
    int i2 = coll.compare(cs1,cs1+s1.size(),cs2,cs2+s2.size());

    int i3 = locale()(s1,s2);          // compare using the current global locale
    int i4 = locale("")(s1,s2);        // compare using my preferred locale
    int i5 = locale(n)(s1,s2);         // compare using the locale named n

    // ...
}
```

Here, *i0==i3*, *i1==i4*, and *i2==i5*. It is not difficult to imagine cases in which *i2*, *i3*, and *i4* differ. Consider this sequence of words from a German dictionary:

Dialekt, *Diät*, *dich*, *dichten*, *Dichtung*

According to convention, nouns (only) are capitalized, but the ordering is not case sensitive.

A case-sensitive German sort would place all words starting with *D* before *d*:

Dialekt, *Diät*, *Dichtung*, *dich*, *dichten*

The *ä* (umlaut *a*) is treated as ''a kind of *a*,'' so it comes before *c*. However, in most common character sets, the numeric value of *ä* is larger than the numeric value of *c*. Consequently, *int*(´*c*´)<*int*(´*ä*´), and the simple default sort based on numeric values gives:

Dialekt, *Dichtung*, *Diät*, *dich*, *dichten*

Writing a compare function that orders this sequence correctly according to the dictionary is an interesting exercise (§D.6[3]).

The *hash*() function calculates a hash value (§17.6.2.3). Obviously, this can be useful for building hash tables.

The *transform*() function produces a string that, when compared to other strings, gives the same result as would a comparison to the argument string. The purpose of *transform*() is to allow optimization of code in which one string is compared to many others. This is useful when implementing a search for one or more strings among a set of strings.

The public *compare*(), *hash*(), and *transform*() functions are implemented by calls to the protected virtual functions *do_compare*(), *do_hash*(), and *do_transform*(), respectively. These ''*do_* functions'' can be overridden in derived classes. This two-function strategy allows the library implementer who writes the non-virtual functions to provide some common functionality for all calls independently of what the user-supplied *do_* function might do.

The use of virtual functions preserves the polymorphic nature of the *facet* but could be costly. To avoid excess function calls, a *locale* can determine the exact *facet* used and cache any values it might need for efficient execution (§D.2.2).

The static member *id* of type *locale*::*id* is used to identify a *facet* (§D.3). The standard functions *has_facet* and *use_facet* depend on the correspondence between *id*s and *facet*s (§D.3.1). Two *facet*s providing exactly the same interface and semantics to *locale* should have the same *id*. For example, *collate*<*char*> and *collate_byname*<*char*> (§D.4.1.1) have the same *id*. Conversely, two *facet*s performing different functions (as far as *locale* is concerned) must have different *id*s. For example, *numpunct*<*char*> and *num_put*<*char*> have different *id*s (§D.4.2).

D.4.1.1 Named Collate

A ***collate_byname*** is a facet that provides a version of ***collate*** for a particular locale named by a constructor string argument:

```
template <class Ch>
class std::collate_byname : public collate<Ch> {
public:
    typedef basic_string<Ch> string_type;

    explicit collate_byname(const char*, size_t r = 0);  // construct from named locale

    // note: no id and no new functions

protected:
    ~collate_byname();  // note: protected destructor

    // override collate<Ch>'s virtual functions:

    int do_compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const;
    string_type do_transform(const Ch* b, const Ch* e) const;
    long do_hash(const Ch* b, const Ch* e) const;
};
```

Thus, a ***collate_byname*** can be used to pick out a ***collate*** from a locale named in the program's execution environment (§D.4). One obvious way of storing facets in an execution environment would be as data in a file. A less flexible alternative would be to represent a facet as program text and data in a ***_byname*** facet.

The ***collate_byname<char>*** class is an example of a ***facet*** that doesn't have its own ***id*** (§D.3). In a ***locale***, ***collate_byname<Ch>*** is interchangeable with ***collate<Ch>***. A ***collate*** and a ***collate_byname*** for the same locale differ only in the extra constructor offered by the ***collate_byname*** and in the semantics provided by the ***collate_byname***.

Note that the ***_byname*** destructor is ***protected***. This implies that you cannot have a ***_byname*** facet as a local variable. For example:

```
void f()
{
    collate_byname<char> my_coll("");  // error: cannot destroy my_coll
    // ...
}
```

This reflects the view that using locales and facets is something that is best done at a fairly high level in a program to affect large bodies of code. An example is setting the global locale (§D.2.3) or imbuing a stream (§21.6.3, §D.1). If necessary, we could derive a class with a public destructor from a ***_byname*** class and create local variables of that class.

D.4.2 Numeric Input and Output

Numeric output is done by a ***num_put*** facet writing into a stream buffer (§21.6.4). Conversely, numeric input is done by a ***num_get*** facet reading from a stream buffer. The format used by ***num_put*** and ***num_get*** is defined by a ''numerical punctuation'' facet, ***numpunct***.

D.4.2.1 Numeric Punctuation

The ***numpunct*** facet defines the I/O format of built-in types, such as ***bool***, ***int***, and ***double***:

```
template <class Ch>
class std::numpunct : public locale::facet {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit numpunct(size_t r = 0);

    Ch decimal_point() const;           // '.' in classic()
    Ch thousands_sep() const;           // ',' in classic()
    string grouping() const;            // "" in classic(), meaning no grouping

    string_type truename() const;       // "true" in classic()
    string_type falsename() const;      // "false" in classic()

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

protected:
    ~numpunct();

    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

The characters of the string returned by ***grouping()*** are read as a sequence of small integer values. Each number specifies a number of digits for a group. Character 0 specifies the rightmost group (the least-significant digits), character 1 the group to the left of that, etc. Thus, `"\004\002\003"` describes a number, such as *123-45-6789* (provided you use ´-´ as the separation character). If necessary, the last number in a grouping pattern is used repeatedly, so `"\003"` is equivalent to `"\003\003\003"`. As the name of the separation character, ***thousands_sep()***, indicates, the most common use of grouping is to make large integers more readable. The ***grouping()*** and ***thousands_sep()*** functions define a format for both input and output of integers. They also define the the format for the integer part of a floating point number, but not for the digits after the ***decimal_point()***.

We can define a new punctuation style by deriving from ***numpunct***. For example, I could define facet ***My_punct*** to write integer values using spaces to group the digits by threes and floating-point values, using a European-style comma as the ''decimal point:''

```
class My_punct : public std::numpunct<char> {
public:
    typedef char char_type;
    typedef string string_type;

    explicit My_punct(size_t r = 0) : std::numpunct<char>(r) { }

protected:
    char do_decimal_point() const { return ´,´; } // comma
    char do_thousands_sep() const { return ´ ´; } // space
    string do_grouping() const { return "\003"; } // 3-digit groups
};
```

```
void f()
{
    cout << setprecision(4) << fixed;
    cout << "style A: " << 12345678 << " *** " << 1234.5678 << '\n';
    locale loc(locale(),new My_punct);
    cout.imbue(loc);
    cout << "style B: " << 12345678 << " *** " << 1234.5678 << '\n';
}
```

This produced:

```
style A: 12345678 *** 1234.5678
style B: 12 345 678 *** 1 234,5678
```

Note that **imbue()** stores a copy of its argument in its stream. Consequently, a stream can rely on an imbued locale even after the original copy of that locale has been destroyed. If an iostream has its **boolalpha** flag set (§21.2.2, §21.4.1), the strings returned by **truename()** and **falsename()** are used to represent **true** and **false**, respectively; otherwise, **1** and **0** are used.

A **_byname** version (§D.4, §D.4.1) of **numpunct** is provided:

```
template<class Ch> class std::numpunct_byname : public numpunct<Ch> { /* ... */ };
```

D.4.2.2 Numeric Output

When writing to a stream buffer (§21.6.4), an **ostream** relies on the **num_put** facet:

```
template<class Ch, class Out = ostreambuf_iterator<Ch> >
class std::num_put : public locale::facet {
public:
    typedef Ch char_type;
    typedef Out iter_type;

    explicit num_put(size_t r = 0);

    // put value "v" to buffer position "b" in stream "s":
    Out put(Out b, ios_base& s, Ch fill, bool v) const;
    Out put(Out b, ios_base& s, Ch fill, long v) const;
    Out put(Out b, ios_base& s, Ch fill, unsigned long v) const;
    Out put(Out b, ios_base& s, Ch fill, double v) const;
    Out put(Out b, ios_base& s, Ch fill, long double v) const;
    Out put(Out b, ios_base& s, Ch fill, const void* v) const;

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

protected:
    ~num_put();

    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

The output iterator (§19.1, §19.2.1) argument, **Out**, identifies where in an **ostream**'s stream buffer (§21.6.4) **put()** places characters representing the numeric value on output. The value of **put()** is

that iterator positioned one past the last character position written.

Note that the default specialization of ***num_put*** (the one where the iterator used to access characters is of type ***ostreambuf_iterator<Ch>***) is part of the standard locales (§D.4). If you want to use another specialization, you'll have to make it yourself. For example:

```
template<class Ch>
class String_numput : public std::num_put<Ch,typename basic_string<Ch>::iterator> {
public:
    String_numput() : std::num_put<Ch,typename basic_string<Ch>::iterator>(1) { }
};

void f(int i, string& s, int pos)       // format i into s starting at pos
{
    String_numput<char> f;
    ios_base& xxx = cout;                   // use cout's formatting rules
    f.put(s.begin()+pos,xxx,' ',i);     // format i into s
}
```

The ***ios_base*** argument is used to get information about formatting state and locale. For example, if padding is needed, the ***fill*** character is used as required by the ***ios_base*** argument. Typically, the stream buffer written to through ***b*** is the buffer associated with an ***ostream*** for which ***s*** is the base. Note that an ***ios_base*** is not a simple object to construct. In particular, it controls many aspects of formatting that must be consistent to achieve acceptable output. Consequently, ***ios_base*** has no public constructor (§21.3.3).

A ***put()*** function also uses its ***ios_base*** argument to get the stream's ***locale()***. That ***locale*** is used to determine punctuation (§D.4.2.1), the alphabetic representation of Booleans, and the conversion to ***Ch***. For example, assuming that ***s*** is ***put()***'s ***ios_base*** argument, we might find code like this in a ***put()*** function:

```
const locale& loc = s.getloc();
// ...
wchar_t w = use_facet< ctype<char> >(loc).widen(c);                 // char to Ch conversion
// ...
string pnt = use_facet< numpunct<char> >(loc).decimal_point();      // default: '.'
// ...
string flse = use_facet< numpunct<char> >(loc).falsename();         // default: "false"
```

A standard facet, such as ***num_put<char>***, is typically used implicitly through a standard I/O stream function. Consequently, most programmers need not know about it. However, the use of such facets by standard library functions is interesting because they show how I/O streams work and how facets can be used. As ever, the standard library provides examples of interesting programming techniques.

Using ***num_put***, the implementer of ***ostream*** might write:

```
template<class Ch, class Tr>
ostream& std::basic_ostream<Ch,Tr>::operator<<(double d)
{
    sentry guard(*this);        // see §21.3.8
    if (!guard) return *this;
```

```
        try {
            if (use_facet< num_put<Ch> >(getloc()).put(*this, *this, this->fill(), d).failed())
                setstate(badbit);
        }
        catch (...) {
            handle_ioexception(*this);
        }
        return *this;
    }
```

A lot is going on here. The sentry ensures that all prefix and suffix operations are performed (§21.3.8). We get the ***ostream***'s ***locale*** by calling its member function ***getloc()*** (§21.7). We extract ***num_put*** from that ***locale*** using ***use_facet*** (§D.3.1). That done, we call the appropriate ***put()*** function to do the real work. An ***ostreambuf_iterator*** can be constructed from an ***ostream*** (§19.2.6), and an ***ostream*** can be implicitly converted to its base class ***ios_base*** (§21.2.1), so the two first arguments to ***put()*** are easily supplied.

A call of ***put()*** returns its output iterator argument. This output iterator is obtained from a ***basic_ostream***, so it is an ***ostreambuf_iterator***. Consequently, ***failed()*** (§19.2.6.1) is available to test for failure and to allow us to set the stream state appropriately.

I did not use ***has_facet***, because the standard facets (§D.4) are guaranteed to be present in every locale. If that guarantee is violated, ***bad_cast*** is thrown (§D.3.1).

The ***put()*** function calls the virtual ***do_put()***. Consequently, user-defined code may be executed, and ***operator<<()*** must be prepared to handle an exception thrown by the overriding ***do_put()***. Also, ***num_put*** may not exist for some character types, so ***use_facet()*** might throw ***std::bad_cast*** (§D.3.1). The behavior of a << for a built-in type, such as ***double***, is defined by the C++ standard. Consequently, the question is not what ***handle_ioexception()*** should do but rather how it should do what the standard prescribes. If ***badbit*** is set in this ***ostream***'s exception state (§21.3.6), the exception is simply rethrown. Otherwise, an exception is handled by setting the stream state and continuing. In either case, ***badbit*** must be set in the stream state (§21.3.3):

```
template<class Ch, class Tr>
void handle_ioexception(std::basic_ostream<Ch,Tr>& s)   // called from catch clause
{
    if (s.exceptions()&ios_base::badbit) {
        try {
            s.setstate(ios_base::badbit);   // might throw basic_ios::failure
        } catch(...) { }
        throw;      // rethrow
    }
    s.setstate(ios_base::badbit);
}
```

The *try-block* is needed because ***setstate()*** might throw ***basic_ios::failure*** (§21.3.3, §21.3.6). However, if ***badbit*** is set in the exception state, ***operator<<()*** must rethrow the exception that caused ***handle_ioexception()*** to be called (rather than simply throwing ***basic_ios::failure***).

The << for a built-in type, such as ***double***, must be implemented by writing directly to a stream buffer. When writing a << for a user-defined type, we can often avoid the resulting complexity by expressing the output of the user-defined type in terms of output of existing types (§D.3.2).

D.4.2.3 Numeric Input

When reading from a stream buffer (§21.6.4), an ***istream*** relies on the ***num_get*** facet:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class std::num_get : public locale::facet {
public:
    typedef Ch char_type;
    typedef In iter_type;

    explicit num_get(size_t r = 0);

    // read [b:e) into v, using formatting rules from s, reporting errors by setting r:
    In get(In b, In e, ios_base& s, ios_base::iostate& r, bool& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, long& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned short& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned int& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned long& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, float& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, double& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, long double& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, void*& v) const;

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

protected:
    ~num_get();

    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

Basically, ***num_get*** is organized like ***num_put*** (§D.4.2.2). Since it reads rather than writes, ***get***( ) needs a pair of input iterators, and the argument designating the target of the read is a reference.

The ***iostate*** variable ***r*** is set to reflect the state of the stream. If a value of the desired type could not be read, ***failbit*** is set in ***r***; if the end of input was reached, ***eofbit*** is set in ***r***. An input operator will use ***r*** to determine how to set the state of its stream. If no error was encountered, the value read is assigned though ***v***; otherwise, ***v*** is left unchanged.

A sentry is used to ensure that the stream's prefix and suffix operations are performed (§21.3.8). In particular, the sentry is used to ensure that we try to read only if the stream is in a good state to start with.

The implementer of ***istream*** might write:

```
template<class Ch, class Tr>
istream& std::basic_istream<Ch, Tr>::operator>>(double& d)
{
    sentry guard(*this);        // see §21.3.8
    if (!guard) return *this;

    iostate state = 0;      // good
    istreambuf_iterator<Ch> eos;
    double dd;
```

```
        try {
                use_facet< num_get<Ch> >(getloc()).get(*this,eos,*this,state,dd);
                if (state==0 || state==eofbit) d = dd; // set value only if get() succeeded
                setstate(state);
        }
        catch (...) {
                handle_ioexception(*this);      // see §D.4.2.2
        }
        return *this;
}
```

Exceptions enabled for the *istream* will be thrown by *setstate()* in case of error (§21.3.6).

By defining a *numpunct*, such as *My_punct* from §D.4.2, we can read using nonstandard punctuation. For example:

```
void f()
{
        cout << "style A: "
        int i1;
        double d1;
        cin >> i1 >> d1;                // read using standard ''12345678'' format

        locale loc(locale::classic(),new My_punct);
        cin.imbue(loc);
        cout << "style B: "
        int i2;
        double d2;
        cin >> i1 >> d2;                // read using the ''12 345 678'' format
}
```

If we want to read really unusual numeric formats, we have to override *do_get()*. For example, we might define a *num_get* that read Roman numerals, such as *XXI* and *MM* (§D.6[15]).

D.4.3 Input and Output of Monetary Values

The formatting of monetary amounts is technically similar to the formatting of ''plain'' numbers (§D.4.2). However, the presentation of monetary amounts is even more sensitive to cultural differences. For example, a negative amount (a loss, a debit), such as *-1.25*, should in some contexts be presented as a (positive) number in parentheses: *(1.25)*. Similarly, color is in some contexts used to ease the recognition of negative amounts.

There is no standard ''money type.'' Instead, the money facets are meant to be used explicitly for numeric values that the programmer knows to represent monetary amounts. For example:

```
class Money { // simple type to hold a monetary amount
        long int amount;
public:
        Money(long int i) : amount(i) { }
        operator long int() const { return amount; }
};
```

```
// ...

void f(long int i)
{
    cout << "value= " << i << " amount= " << Money(i) << endl;
}
```

The task of the monetary facets is to make it reasonably easy to write an output operator for ***Money*** so that the amount is printed according to local convention (see §D.4.3.2). The output would vary depending on ***cout***'s locale. Possible outputs are:

```
value= 1234567 amount= $12345.67
value= 1234567 amount= 12345,67 DKK
value= -1234567 amount= $-12345.67
value= -1234567 amount= -$12345.67
value= -1234567 amount= (CHF12345,67)
```

For money, accuracy to the smallest currency unit is usually considered essential. Consequently, I adopted the common convention of having the integer value represent the number of cents (pence, øre, fils, cents, etc.) rather than the number of dollars (pounds, kroner, dinar, euro, etc.). This convention is supported by ***moneypunct***'s ***frac_digits***() function (§D.4.3.1). Similarly, the appearance of the ''decimal point'' is defined by ***decimal_point***().

The facets ***money_get*** and ***money_put*** provide functions that perform I/O based on the format defined by the ***money_base*** facet.

A simple ***Money*** type can be used simply to control I/O formats or to hold monetary values. In the former case, we cast values of (other) types used to hold monetary amounts to ***Money*** before writing, and we read into ***Money*** variables before converting them to other types. It is less error prone to consistently hold monetary amounts in a ***Money*** type; that way, we cannot forget to cast a value to ***Money*** before writing it, and we don't get input errors by trying to read monetary values in locale-insensitive ways. However, it may be infeasible to introduce a ***Money*** type into a system that wasn't designed for that. In such cases, applying ***Money*** conversions (casts) to read and write operations is necessary.

D.4.3.1 Money Punctuation

The facet controlling the presentation of monetary amounts, ***moneypunct***, naturally resembles the facet for controlling plain numbers, ***numpunct*** (§D.4.2.1):

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

```
template <class Ch, bool International = false>
class std::moneypunct : public locale::facet, public money_base {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit moneypunct(size_t r = 0);

    Ch decimal_point() const;           // '.' in classic()
    Ch thousands_sep() const;           // ',' in classic()
    string grouping() const;            // "" in classic(), meaning "no grouping"

    string_type curr_symbol() const;    // "$" in classic()
    string_type positive_sign() const;  // "" in classic()
    string_type negative_sign() const;  // "-" in classic()

    int frac_digits() const;            // number of digits after the decimal point; 2 in classic()
    pattern pos_format() const;         // { symbol, sign, none, value } in classic()
    pattern neg_format() const;         // { symbol, sign, none, value } in classic()

    static const bool intl = International;     // use international monetary formats

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

protected:
    ~moneypunct();

    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

The facilities offered by ***moneypunct*** are intended primarily for use by implementers of ***money_put*** and ***money_get*** facets (§D.4.3.2, §D.4.3.3).

The ***decimal_point***(), ***thousands_sep***(), and ***grouping***() members behave as their equivalents in ***numpunct***.

The ***curr_symbol***(), ***positive_sign***(), and ***negative_sign***() members return the string to be used to represent the currency symbol (for example, $, ¥, ***FRF***, ***DKK***), the plus sign, and the minus sign, respectively. If the ***International*** template argument was ***true***, the ***intl*** member will also be ***true***, and ''international'' representations of the currency symbols will be used. Such an ''international'' representation is a four-character string. For example:

```
"USD "
"DKK "
"EUR "
```

The last character is a terminating zero. The three-letter currency identifier is defined by the ISO-4217 standard. When ***International*** is ***false***, a ''local'' currency symbol, such as $, £, and ¥, can be used.

A ***pattern*** returned by ***pos_format***() or ***neg_format***() is four ***part***s defining the sequence in which the numeric value, the currency symbol, the sign symbol, and whitespace occur. Most common formats are trivially represented using this simple notion of a pattern. For example:

```
+$123.45      // { sign, symbol, space, value } where positive_sign() returns "+"
$+123.45      // { symbol, sign, value, none } where positive_sign() returns "+"
$123.45       // { symbol, sign, value, none } where positive_sign() returns ""
$123.45-      // { symbol, value, sign, none }
-123.45 DKK   // { sign, value, space, symbol }
($123.45)     // { sign, symbol, value, none } where negative_sign() returns "()"
(123.45DKK)   // { sign, value, symbol, none } where negative_sign() returns "()"
```

Representing a negative number using parentheses is achieved by having ***negative_sign***`()` return a string containing the two characters `()`. The first character of a sign string is placed where ***sign*** is found in the pattern, and the rest of the sign string is placed after all other parts of the pattern. The most common use of this facility is to represent the financial community's convention of using parentheses for negative amounts, but other uses are possible. For example:

```
-$123.45          // { sign, symbol, value, none } where negative_sign() returns "–"
*$123.45 silly    // { sign, symbol, value, none } where negative_sign() returns "* silly"
```

The values ***sign***, ***value***, and ***symbol*** must each appear exactly once in a pattern. The remaining value can be either ***space*** or ***none***. Where ***space*** appears, at least one and possibly more white-space characters may appear in the representation. Where ***none*** appears, except at the end of a pattern, zero or more whitespace characters may appear in the representation.

Note that these strict rules ban some apparently reasonable patterns:

```
pattern pat = { sign, value, none, none };      // error: no symbol
```

The ***frac_digits***`()` function indicates where the ***decimal_point***`()` is placed. Often, monetary amounts are represented in the smallest currency unit (§D.4.3). This unit is typically one hundredth of the major unit (for example, a ¢ is one hundredth of a $), so ***frac_digits***`()` is often *2*.

Here is a simple format defined as a facet:

```
class My_money_io : public moneypunct<char,true> {
public:
    explicit My_money_io(size_t r = 0) :moneypunct<char,true>(r) { }

    char_type do_decimal_point() const { return '.'; }
    char_type do_thousands_sep() const { return ','; }
    string do_grouping() const { return "\003\003\003"; }

    string_type do_curr_symbol() const { return "USD "; }
    string_type do_positive_sign() const { return ""; }
    string_type do_negative_sign() const { return "()"; }

    int do_frac_digits() const { return 2; }    // two digits after decimal point

    pattern do_pos_format() const
    {
        static pattern pat = { sign, symbol, value, none };
        return pat;
    }
```

```
        pattern do_neg_format() const
        {
            static pattern pat = { sign, symbol, value, none };
            return pat;
        }
};
```

This facet is used in the ***Money*** input and output operations defined in §D.4.3.2 and §D.4.3.3.

A ***_byname*** version (§D.4, §D.4.1) of ***moneypunct*** is provided:

```
template <class Ch, bool Intl = false>
class std::moneypunct_byname : public moneypunct<Ch, Intl> { /* ... */ };
```

D.4.3.2 Money Output

The ***money_put*** facet writes monetary amounts according to the format specified by ***moneypunct***. Specifically, ***money_put*** provides ***put()*** functions that place a suitably formatted character representation into the stream buffer of a stream:

```
template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::money_put : public locale::facet {
public:
    typedef Ch char_type;
    typedef Out iter_type;
    typedef basic_string<Ch> string_type;

    explicit money_put(size_t r = 0);

    // put value "v" into buffer position "b":
    Out put(Out b, bool intl, ios_base& s, Ch fill, long double v) const;
    Out put(Out b, bool intl, ios_base& s, Ch fill, const string_type& v) const;

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

protected:
    ~money_put();

    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

The ***b***, ***s***, ***fill***, and ***v*** arguments are used as for ***num_put***'s ***put()*** functions (§D.4.2.2). The ***intl*** argument indicates whether a standard four-character ''international'' currency symbol or a ''local'' symbol is used (§D.4.3.1).

Given ***money_put***, we can define an output operator for ***Money*** (§D.4.3):

```
ostream& operator<<(ostream& s, Money m)
{
    ostream::sentry guard(s);          // see §21.3.8
    if (!guard) return s;
```

```
        try {
                const money_put<char>& f = use_facet< money_put<char> >(s.getloc());
                if (m==static_cast<long double>(m)) {  // m can be represented as a long double
                        if (f.put(s, true, s, s.fill(), m).failed()) s.setstate(ios_base::badbit);
                }
                else {
                        ostringstream v;
                        v << m;             // convert to string representation
                        if (f.put(s, true, s, s.fill(), v.str()).failed()) s.setstate(ios_base::badbit);
                }
        }
        catch (...) {
                handle_ioexception(s);      // see §D.4.2.2
        }
        return s;
}
```

If a ***long double*** doesn't have sufficient precision to represent the monetary value exactly, I convert the value to its string representation and output that using the ***put***() that takes a ***string***.

D.4.3.3 Money Input

The ***money_get*** facet reads monetary amounts according to the format specified by ***moneypunct***. Specifically, ***money_get*** provides ***get***() functions that extract a suitably formatted character representation from the stream buffer of a stream:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class std::money_get : public locale::facet {
public:
        typedef Ch char_type;
        typedef In iter_type;
        typedef basic_string<Ch> string_type;

        explicit money_get(size_t r = 0);

        // read [b:e) into v, using formatting rules from s, reporting errors by setting r:
        In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, long double& v) const;
        In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, string_type& v) const;

        static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)
protected:
        ~money_get();

        // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

The ***b***, ***e***, ***s***, ***fill***, and ***v*** arguments are used as for ***num_get***'s ***get***() functions (§D.4.2.3). The ***intl*** argument indicates whether a standard four-character ''international'' currency symbol or a ''local'' symbol is used (§D.4.3.1).

A well-defined pair of *money_get* and *money_put* facets will provide output in a form that can be read back in without errors or loss of information. For example:

```
int main()
{
    Money m;
    while (cin>>m) cout << m << "\n";
}
```

The output of this simple program should be acceptable as its input. Furthermore, the output produced by a second run given the output from a first run should be identical to its input.

A plausible input operator for *Money* would be:

```
istream& operator>>(istream& s, Money& m)
{
    istream::sentry guard(s);        // see §21.3.8
    if (guard) try {
        ios_base::iostate state = 0;      // good
        istreambuf_iterator<char> eos;
        string str;

        use_facet< money_get<char> >(s.getloc()).get(s,eos,true,state,str);

        if (state==0 || state==ios_base::eofbit) { // set value only if get() succeeded
            long int i = strtol(str.c_str(),0,0); // for strtol(), see §20.4.1
            if (errno==ERANGE)
                state |= ios_base::failbit;
            else
                m = i;      // set value only if conversion to long int succeeded
            s.setstate(state);
        }
    }
    catch (...) {
        handle_ioexception(s);     // see §D.4.2.2
    }
    return s;
}
```

I use the *get()* that reads into a *string* because reading into a *double* and then converting to a *long int* could lead to loss of precision.

D.4.4 Date and Time Input and Output

Unfortunately, the C++ standard library does not provide a proper *date* type. However, from the C standard library, it inherits low-level facilities for dealing with dates and time intervals. These C facilities are the basis for C++'s facilities for dealing with time in a system-independent manner.

The following sections demonstrate how the presentation of date and time-of-day information can be made *locale* sensitive. In addition, they provide an example of how a user-defined type (*Date*) can fit into the framework provided by *iostream* (Chapter 21) and *locale* (§D.2). The implementation of *Date* shows techniques that are useful for dealing with time if you don't have a *Date* type available.

D.4.4.1 Clocks and Timers

At the lowest level, most systems have a fine-grained timer. The standard library provides a function ***clock***( ) that returns an implementation-defined arithmetic type ***clock_t***. The result of ***clock***( ) can be calibrated by using the ***CLOCKS_PER_SEC*** macro. If you don't have access to a reliable timing utility, you might measure a loop like this:

```
int main(int argc, char* argv[])    // §6.1.7
{
    int n = atoi(argv[1]);          // §20.4.1

    clock_t t1 = clock();
    if (t1 == clock_t(-1)) {        // clock_t(-1) means "clock() didn't work"
        cerr << "sorry, no clock\n";
        exit(1);
    }

    for (int i = 0; i<n; i++) do_something(); // timing loop

    clock_t t2 = clock();
    if (t2 == clock_t(-1)) {
        cerr << "sorry, clock overflow\n";
        exit(2);
    }
    cout << "do_something() " << n << " times took "
        << double(t2-t1)/CLOCKS_PER_SEC << " seconds"
        << " (measurement granularity: " << CLOCKS_PER_SEC << " of a second)\n";
}
```

The explicit conversion ***double***(*t2-t1*) before dividing is necessary because ***clock_t*** might be an integer. Exactly when the ***clock***( ) starts running is implementation defined; ***clock***( ) is meant to measure time intervals within a single run of a program. For values *t1* and *t2* returned by ***clock***( ), ***double***(*t2-t1*)*/CLOCKS_PER_SEC* is the system's best approximation of the time in seconds between the two calls.

If ***clock***( ) isn't provided for a processor or if a time interval was too long to measure, ***clock***( ) returns ***clock_t***(*-1*).

The ***clock***( ) function is meant to measure intervals from a fraction of a second to a few seconds. For example, if ***clock_t*** is a 32-bit signed ***int*** and ***CLOCKS_PER_SEC*** is 1,000,000 , we can use ***clock***( ) to measure from 0 to just over 2,000 seconds (about half an hour) in microseconds.

Please note that getting meaningful measurements of a program can be tricky. Other programs running on a machine may severely affect the time used by a run, cache and pipelining effects are difficult to predict, and algorithms may have surprising dependencies on data. If you try to time something, make several runs and reject the results as flawed if the run times vary significantly.

To cope with longer time intervals and with calendar time, the standard library provides ***time_t*** for representing a point in time and a structure ***tm*** for separating a point in time into its conventional parts:

```
typedef implementation_defined time_t;      // implementation-defined arithmetic type (§4.1.1)
                                            // capable of representing a period of time,
                                            // often, a 32-bit integer

struct tm {
    int tm_sec;     // second of minute [0,61]; 60 and 61 to represent leap seconds
    int tm_min;     // minute of hour [0,59]
    int tm_hour;    // hour of day [0,23]
    int tm_mday;    // day of month [1,31]
    int tm_mon;     // month of year [0,11]; 0 means January (note: not [1:12])
    int tm_year;    // year since 1900; 0 means year 1900, and 102 means 2002
    int tm_wday;    // days since Sunday [0,6]; 0 means Sunday
    int tm_yday;    // days since January 1 [0,365]; 0 means January 1
    int tm_isdst;   // hours of daylight savings time
};
```

Note that the standard guarantees only that ***tm*** has the ***int*** members mentioned here. The standard does not guarantee that the members appear in this order or that there are no other fields.

The ***time_t*** and ***tm*** types and the basic facilities for using them are presented in ***<ctime>*** and ***<time.h>***. For example::

```
clock_t clock();                            // number of clock ticks since the start of the program

time_t time(time_t* pt);                    // current calendar time
double difftime(time_t t2, time_t t1);      // t2–t1 in seconds

tm* localtime(const time_t* pt);            // local time for the *pt
tm* gmtime(const time_t* pt);               // Greenwich Mean Time (GMT) tm for *pt, or 0
                                            // (officially called Coordinated Universal Time, UTC)

time_t mktime(tm* ptm);                     // time_t for *ptm, or time_t(-1)

char* asctime(const tm* ptm);               // C-style string representation for *ptm
                                            // for example, "Sun Sep 16 01:03:52 1973\n"
char* ctime(const time_t* t) { return asctime(localtime(t)); }
```

Beware: both ***localtime()*** and ***gmtime()*** return a ***tm**** to a statically allocated object; a subsequent call of that function will change the value of that object. Either use such a return value immediately, or copy the ***tm*** into storage that you control. Similarly, ***asctime()*** returns a pointer to a statically allocated character array.

A ***tm*** can represent dates in a range of at least tens of thousands of years (about [-32000,32000] for a minimally sized ***int***). However, ***time_t*** is most often a (signed) 32-bit ***long int***. Counting seconds, this makes ***time_t*** capable of representing a range just over 68 years on each side of a base year. This base year is most commonly 1970, with the exact base time being 0:00 of January 1 GMT (UTC). If ***time_t*** is a 32-bit signed integer, we'll run out of ''time'' in 2038 unless we upgrade ***time_t*** to a larger integer type, as is already done on some systems.

The ***time_t*** mechanism is meant primarily for representing ''near current time.'' Thus, we should not expect ***time_t*** to be able to represent dates outside the [1902,2038] range. Worse, not all implementations of the functions dealing with time handle negative values in the same way. For portability, a value that needs to be represented as both a ***tm*** and a ***time_t*** should be in the

[1970,2038] range. People who want to represent dates outside the 1970 to 2038 time frame must devise some additional mechanism to do so.

One consequence of this is that ***mktime()*** can fail. If the argument for ***mktime()*** cannot be represented as a ***time_t***, the error indicator ***time_t(-1)*** is returned.

If we have a long-running program, we might time it like this:

```
int main(int argc, char* argv[])    // §6.1.7
{
    time_t t1 = time(0);
    do_a_lot(argc,argv);
    time_t t2 = time(0);
    double d = difftime(t2,t1);
    cout << "do_a_lot() took" << d << " seconds\n";
}
```

If the argument to ***time()*** is not ***0***, the resulting time is also assigned to the ***time_t*** pointed to. If the calendar time is not available (say, on a specialized processor), the value ***time_t(-1)*** is returned. We could cautiously try to find today's date like this:

```
int main()
{
    time_t t;

    if (time(&t) == time_t(-1)) {   // time_t(–1) means ''time() didn't work''
        cerr << "Bad time\n";
        exit(1);
    }

    tm* gt = gmtime(&t);
    cout << gt->tm_mon+1 << '/' << gt->tm_mday << '/' << 1900+gt->tm_year << endl;
}
```

D.4.4.2 A Date Class

As mentioned in §10.3, it is unlikely that a single ***Date*** type can serve all purposes. The uses of date information dictate a variety of representations, and calendar information before the 19th century is very dependent on historical vagaries. However, as an example, we could define a ***Date*** type along the lines from §10.3, using ***time_t*** as the implementation:

```
class Date {
public:
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    class Bad_date {};

    Date(int dd, Month mm, int yy);
    Date();

    friend ostream& operator<<(ostream& s, const Date& d);
```

```
        // ...
    private:
        time_t d;   // standard date and time representation
    };

    Date::Date(int dd, Month mm, int yy)
    {
        tm x = { 0 };
        if (dd<0 || 31<dd) throw Bad_date();      // oversimplified: see §10.3.1
        x.tm_mday = dd;
        if (mm<jan || dec<mm) throw Bad_date();
        x.tm_mon = mm-1;          // tm_mon is zero based
        x.tm_year = yy-1900;      // tm_year is 1900 based
        d = mktime(&x);
    }

    Date::Date()
    {
        d = time(0);      // default Date: today
        if (d == time_t(-1)) throw Bad_date();
    }
```

The task here is to define locale-sensitive implementations for *Date* << and >>.

D.4.4.3 Date and Time Output

Like *num_put* (§D.4.2), *time_put* provides *put*() functions for writing to buffers through iterators:

```
    template <class Ch, class Out = ostreambuf_iterator<Ch> >
    class std::time_put : public locale::facet {
    public:
        typedef Ch char_type;
        typedef Out iter_type;

        explicit time_put(size_t r = 0);

        // put t into s's stream buffer through b, using format fmt:
        Out put(Out b, ios_base& s, Ch fill, const tm* t,
                                        const Ch* fmt_b, const Ch* fmt_e) const;

        Out put(Out b, ios_base& s, Ch fill, const tm* t, char fmt, char mod = 0) const
            { return do_put(b,s,fill,t,fmt,mod); }

        static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

    protected:
        ~time_put();

        virtual Out do_put(Out, ios_base&, Ch, const tm*, char, char) const;
    };
```

A call *put*(*b*,*s*,*fill*,*t*,*fmt_b*,*fmt_e*) places the date information from *t* into *s*'s stream buffer through *b*. The *fill* character is used where needed for padding. The output format is specified by a *printf*()-like format string [*fmt_b*,*fmt_e*). The *printf*-like (§21.8) format is used to produce an

actual output and may contain the following special-purpose format specifiers:

%a	abbreviated weekday name (e.g., Sat)
%A	full weekday name (e.g., Saturday)
%b	abbreviated month name (e.g., Feb)
%B	full month name (e.g., February)
%c	date and time (e.g., Sat Feb 06 21:46:05 1999)
%d	day of month [01,31] (e.g., 06)
%H	24-hour clock hour [00,23] (e.g., 21)
%I	12-hour clock hour [01,12] (e.g., 09)
%j	day of year [001,366] (e.g., 037)
%m	month of year [01,12] (e.g., 02)
%M	minute of hour [00,59] (e.g., 48)
%p	a.m./p.m. indicator for 12-hour clock (e.g., PM)
%S	second of minute [00,61] (e.g., 40)
%U	week of year [00,53] starting with Sunday (e.g., 05); the first Sunday starts week 1
%w	day of week [0,6]; 0 means Sunday (e.g., 6)
%W	week of year [00,53] starting with Monday (e.g., 05); the first Monday starts week 1
%x	date (e.g., 02/06/99)
%X	time (e.g., 21:48:40)
%y	year without century [00,99] (e.g., 99)
%Y	year (e.g., 1999)
%Z	time zone indicator (e.g., EST) if the time zone is known

This long list of very specialized formatting rules could be used as an argument for the use of extensible I/O systems. However, as with most specialized notations, it is adequate for its task and often even convenient.

In addition to these formatting directives, most implementations support ''modifiers,'' such as an integer specifying a field width (§21.8), *%10X*. Modifiers for the time-and-date formats are not part of the C++ standard, but some platform standards, such as POSIX, require them. Consequently, modifiers can be difficult to avoid even if their use isn't perfectly portable.

The *sprintf*-like (§21.8) function *strftime*( ) from <*ctime*> or <*time.h*> produces output using the time and date format directives:

```
size_t strftime(char* s, size_t max, const char* format, const tm* tmp);
```

This function places a maximum of *max* characters from **tmp* and the *format* into **s* according the *format*. For example:

```
int main()
{
    const int max = 20;  // sloppy: hope strftime() will never produce more than 20 characters
    char buf[max];
    time_t t = time(0);
    strftime(buf,max,"%A\n",localtime(&t));
    cout << buf;
}
```

On a Wednesday, this will print *Wednesday* in the default *classic*() locale (§D.2.3) and *onsdag* in a Danish locale.

Characters that are not part of a format specified, such as the newline in the example, are simply copied into the first argument (*s*).

When *put*() identifies a format character *f* (and optional modifier character *m*), it calls the virtual *do_put*() to do the actual formatting: *do_put*(*b*,*s*,*fill*,*t*,*f*,*m*).

A call *put*(*b*,*s*,*fill*,*t*,*f*,*m*) is a simplified form of *put*(), where a format character (*f*) and a modifier character (*m*) are explicitly provided. Thus,

```
const char fmt[] = "%10X";
put(b,s,fill,t,fmt,fmt+sizeof(fmt));
```

can be abbreviated to

```
put(b,s,fill,t,'X',10);
```

If a format contains multibyte characters, it must both begin and end in the default state (§D.4.6).

We can use *put*() to implement a *locale*-sensitive output operator for *Date*:

```
ostream& operator<<(ostream& s, const Date& d)
{
    ostream::sentry guard(s);          // see §21.3.8
    if (!guard) return s;

    tm* tmp = localtime(&d.d);
    try {
        if (use_facet< time_put<char> >(s.getloc()).put(s,s,s.fill(),tmp,'x').failed())
            s.setstate(ios_base::failbit);
    }
    catch (...) {
        handle_ioexception(s);     // see §D.4.2.2
    }
    return s;
}
```

Since there is no standard *Date* type, there is no default layout for date I/O. Here, I specified the %*x* format by passing the character 'x' as the format character. Because the %*x* format is the default for *get_time*() (§D.4.4.4), that is probably as close to a standard as one can get. See §D.4.4.5 for an example of how to use alternative formats.

A *_byname* version (§D.4, §D.4.1) of *time_put* is also provided:

```
template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::time_put_byname : public time_put<Ch,Out> { /* ... */ };
```

D.4.4.4 Date and Time Input

As ever, input is trickier than output. When we write code to output a value, we often have a choice among different formats. In addition, when we write input code, we must deal with errors and sometimes the possibility of several alternative formats.

The *time_get* facet implements input of time and date. The idea is that *time_get* of a *locale* can

read the times and dates produced by the ***locale***'s ***time_put***. However, there are no standard ***date*** and ***time*** classes, so a programmer can use a locale to produce output according to a variety of formats. For example, the following representations could all be produced by using a single output statement, using ***time_put*** (§D.4.4.5) from different locales:

```
January 15th 1999
Thursday 15th January 1999
15 Jan 1999AD
Thurs 15/1/99
```

The C++ standard encourages implementers of ***time_get*** to accept dates and time formats as specified by POSIX and other standards. The problem is that it is difficult to standardize the intent to read dates and times in whatever format is conventional in a given culture. It is wise to experiment to see what a given locale provides (§D.6[8]). If a format isn't accepted, a programmer can provide a suitable alternative ***time_get*** facet.

The standard time input ***facet***, ***time_get***, is derived from ***time_base***:

```
struct std::time_base {
    enum dateorder {
        no_order, // no order, possibly more elements (such as day of week)
        dmy,      // day before month before year
        mdy,      // month before day before year
        ymd,      // year before month before day
        ydm       // year before day before month
    };
};
```

An implementer can use this enumeration to simplify the parsing on date formats.

Like ***num_get***, ***time_get*** accesses its buffer through a pair of input iterators:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class time_get : public locale::facet, public time_base {
public:
    typedef Ch char_type;
    typedef In iter_type;

    explicit time_get(size_t r = 0);

    dateorder date_order() const { return do_date_order(); }

    // read [b,e) into d, using formatting rules from s, reporting errors by setting r:
    In get_time(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_year(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_weekday(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_monthname(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;

    static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)
protected:
    ~time_get();
    // virtual ''do_'' functions for public functions (see §D.4.1)
};
```

The *get_time*( ) function calls *do_get_time*( ). The default *get_time*( ) reads time as produced by the *locale*'s *time_put*::*put*( ), using the *%X* format (§D.4.4). Similarly, the *get_date*( ) function calls *do_get_date*( ). The default *get_date*( ) reads a date as produced by the *locale*'s *time_put*::*put*( ), using the *%x* format (§D.4.4).

Thus, the simplest input operator for *Date*s is something like this:

```
istream& operator>>(istream& s, Date& d)
{
    istream::sentry guard(s);        // see §21.3.8
    if (!guard) return s;

    ios_base::iostate res = 0;
    tm x = { 0 };
    istreambuf_iterator<char,char_traits<char> > end;
    try {
        use_facet< time_get<char> >(s.getloc()).get_date(s,end,s,res,&x);
        if (res==0 || res==ios_base::eofbit)
            d = Date(x.tm_mday,Date::Month(x.tm_mon+1),x.tm_year+1900);
        else
            s.setstate(res);
    }
    catch (...) {
        handle_ioexception(s);       // see §D.4.2.2
    }

    return s;
}
```

The call *get_date*(*s*,*end*,*s*,*res*,&*x*) relies on two implicit conversions from *istream*: As the first argument, *s* is used to construct an *istreambuf_iterator*. As third argument, *s* is converted to the *istream* base class *ios_base*.

This input operator will work correctly for dates in the range that can be represented by *time_t*.

A trivial test case would be:

```
int main()
try {
    Date today;
    cout << today << endl;           // write using %x format
    Date d(12, Date::may, 1998);

    cout << d << endl;
    Date dd;
    while (cin >> dd) cout << dd << endl;      // read dates produced by %x format
}
catch (Date::Bad_date) {
    cout << "exit: bad date caught\n";
}
```

A *_byname* version (§D.4, §D.4.1) of *time_get* is also provided:

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class std::time_get_byname : public time_get<Ch,In> { /* ... */ };
```

D.4.4.5 A More Flexible Date Class

If you tried to use the ***Date*** class from §D.4.4.2 with the I/O from §D.4.4.3 and §D.4.4.4, you'd soon find it restrictive:

[1] It can handle only dates that can be represented by a ***time_t***; that typically means in the [1970,2038] range.
[2] It accepts dates only in the standard format – whatever that might be.
[3] Its reporting of input errors is unacceptable.
[4] It supports only streams of ***char*** – not streams of arbitrary character types.

A more interesting and more useful input operator would accept a wider range of dates, recognize a few common formats, and reliably report errors in a useful form. To do this, we must depart from the ***time_t*** representation:

```
class Date {
public:
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    struct Bad_date {
        const char* why;
        Bad_date(const char* p) : why(p) { }
    };

    Date(int dd, Month mm, int yy, int day_of_week = 0);
    Date();

    void make_tm(tm* t) const;      // place tm representation of Date in *t
    time_t make_time_t() const;     // return time_t representation of Date

    int year() const { return y; }
    Month month() const { return m; }
    int day() const { return d; }

    // ...
private:
    char d;
    Month m;
    int y;
};
```

For simplicity, I reverted to the (d,m,y) representation (§10.2).

The constructor might be defined like this:

```
Date::Date(int dd, Month mm, int yy, int day_of_week)
    :d(dd), m(mm), y(yy)
{
    if (d==0 && m==Month(0) && y==0) return;      // Date(0,0,0) is the "null date"
    if (mm<jan || dec<mm) throw Bad_date("bad month");
```

```
    if (dd<1 || 31<dd)   // oversimplified; see §10.3.1
        throw Bad_date("bad day of month");
    if (day_of_week && day_in_week(yy,mm,dd)!=day_of_week)
        throw Bad_date("bad day of week");
}

Date::Date() :d(0), m(0), y(0) { } // a "null date"
```

The ***day_in_week()*** calculation is nontrivial and immaterial to the ***locale*** mechanisms, so I have left it out. If you need one, your system will have one somewhere.

Comparison operations are always useful for types such as ***Date***:

```
bool operator==(const Date& x, const Date& y)
{
    return x.year()==y.year() && x.month()==y.month() && x.day()==y.day();
}

bool operator!=(const Date& x, const Date& y)
{
    return !(x==y);
}
```

Having departed from the standard ***tm*** and ***time_t*** formats, we need conversion functions to cooperate with software that expects those types:

```
void Date::make_tm(tm* p) const      // put date into *p
{
    tm x = { 0 };
    *p = x;
    p->tm_year = y-1900;
    p->tm_mday = d;
    p->tm_mon = m-1;
}

time_t Date::make_time_t() const
{
    if (y<1970 || 2038<y)      // oversimplified
        throw Bad_date("date out of range for time_t");
    tm x;
    make_tm(&x);
    return mktime(&x);
}
```

D.4.4.6 Specifying a *Date* Format

C++ doesn't define a standard output format for dates (***%x*** is as close as we get; §D.4.4.3). However, even if a standard format existed, we would probably want to be able to use alternatives. This could be done by providing a ''default format'' and a way of changing it. For example:

```
class Date_format {
    static char fmt[];          // default format
    const char* curr;           // current format
    const char* curr_end;
public:
    Date_format() :curr(fmt), curr_end(fmt+strlen(fmt)) { }

    const char* begin() const { return curr; }
    const char* end() const { return curr_end; }

    void set(const char* p, const char* q) { curr=p; curr_end=q; }
    void set(const char* p) { curr=p; curr_end=curr+strlen(p); }

    static const char* default_fmt() { return fmt; }
};

const char Date_format::fmt[] = "%A, %B %d, %Y";  // e.g., Friday, February 5, 1999

Date_format date_fmt;
```

To be able to use that ***strftime()*** format (§D.4.4.3), I have refrained from parameterizing the ***Date_format*** class on the character type used. This implies that this solution allows only date notations for which the format can be expressed as a ***char[]***. I also used a global format object (***date_fmt***) to provide a default ***Date*** format. Since the value of ***date_fmt*** can be changed, this provides a crude way of controlling ***Date*** formatting, similar to the way ***global()*** (§D.2.3) can be used to control formatting.

A more general solution is to add ***Date_in*** and ***Date_out*** facets to control reading and writing from a stream. That approach is presented in §D.4.4.7.

Given ***Date_format***, ***Date::operator<<()*** can be written like this:

```
template<class Ch, class Tr>
basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>& s, const Date& d)
// write according to user-specified format
{
    typename basic_ostream<Ch,Tr>::sentry guard(s);  // see §21.3.8
    if (!guard) return s;

    tm t;
    d.make_tm(&t);
    try {
        const time_put<Ch>& f = use_facet< time_put<Ch> >(s.getloc());
        if (f.put(s,s,s.fill(),&t,date_fmt.begin(),date_fmt.end()).failed())
            s.setstate(ios_base::failbit);
    }
    catch (...) {
        handle_ioexception(s);      // see §D.4.2.2
    }
    return s;
}
```

I could have used ***has_facet*** to verify that ***s***'s locale had a ***time_put<Ch>*** facet. However, here it seemed simpler to handle that problem by catching any exception thrown by ***use_facet***.

Here is a simple test program that controls the output format through *date_fmt*:

```
int main()
try {

    while (cin >> dd && dd != Date()) cout << dd << endl;      // write using default date_fmt

    date_fmt.set("%Y/%m/%d");

    while (cin >> dd && dd != Date()) cout << dd << endl;      // write using "%Y/%m/%d"
}
catch (Date::Bad_date e) {
    cout << "bad date caught: " << e.why << endl;
}
```

D.4.4.7 A *Date* Input Facet

As ever, input is a bit more difficult than output. However, because the interface to low-level input is fixed by *get_date()* and because the *operator>>()* defined for *Date* in §D.4.4.4 didn't directly access the representation of a *Date*, we could use that *operator>>()* unchanged. Here is a templatized version to match the *operator<<()* from §D.4.4.6:

```
template<class Ch, class Tr>
istream<Ch,Tr>& operator>>(istream<Ch,Tr>& s, Date& d)
{
    typename istream<Ch,Tr>::sentry guard(s);
    if (guard) try {
        ios_base::iostate res = 0;
        tm x = { 0 };
        istreambuf_iterator<Ch,Tr> end;

        use_facet< time_get<Ch> >(s.getloc()).get_date(s,end,s,res,&x);

        if (res==0 || res==ios_base::eofbit)
            d = Date(x.tm_mday,Date::Month(x.tm_mon+1),x.tm_year+1900,x.tm_wday);
        else
            s.setstate(res);
    }
    catch (...) {
        handle_ioexception(s);      // see §D.4.2.2
    }
    return s;
}
```

This *Date* input operator calls *get_date()* from the *istream*'s *time_get* facet (§D.4.4.4). Therefore, we can provide a different and more flexible form of input by defining a new facet derived from *time_get*:

```
template<class Ch, class In = istreambuf_iterator<Ch> >
class Date_in : public std::time_get<Ch,In> {
public:
    Date_in(size_t r = 0) : std::time_get<Ch>(r) { }
```

```
protected:
    In do_get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* tmp) const;

private:
    enum Vtype { novalue, unknown, dayofweek, month };
    In getval(In b, In e, ios_base& s, ios_base::iostate& r, int* v, Vtype* res) const;
};
```

The *getval()* needs to read a year, a month, a day of the month, and optionally a day of the week and compose the result into a *tm*.

The names of the months and the names of the days of the week are locale specific. Consequently, we can't mention them directly in our input function. Instead, we recognize months and days by calling the functions that *time_get* provides for that: *get_monthname()* and *get_weekday()* (§D.4.4.4).

The year, the day of the month, and possibly the month are represented as integers. Unfortunately, a number does not indicate whether it denotes a day or a month, or whatever. For example, *7* could denote July, day 7 of a month, or even the year 2007. The real purpose of *time_get*'s *date_order()* is to resolve such ambiguities.

The strategy of *Date_in* is to read values, classify them, and then use *date_order()* to see whether (or how) the values entered make sense. The private *getval()* function does the actual reading from the stream buffer and the initial classification:

```
template<class Ch, class In>
In Date_in<Ch,In>::getval(In b, In e,
                          ios_base& s, ios_base::iostate& r, int* v, Vtype* res) const
    // read part of Date: number, day_of_week, or month. Skip whitespace and punctuation.
{
    const ctype<Ch>& ct = use_facet< ctype<Ch> >(s.getloc()); // ctype is defined in §D.4.5
    Ch c;

    *res = novalue; // no value found

    for (;;) {// skip whitespace and punctuation
        if (b == e) return e;
        c = *b;
        if (!(ct.is(ctype_base::space,c) || ct.is(ctype_base::punct,c))) break;
        ++b;
    }

    if (ct.is(ctype_base::digit,c)) {    // read integer without regard for numpunct
        int i = 0;

        do { // turn digit from arbitrary character set into decimal value:
            static char const digits[] = "0123456789";
            i = i*10 + find(digits,digits+10,ct.narrow(c,'?'))-digits;
            c = *++b;
        } while (ct.is(ctype_base::digit,c));
```

```
            *v = i;
            *res = unknown;        // an integer, but we don't know what it represents
            return b;
        }

        if (ct.is(ctype_base::alpha,c)) {   // look for name of month or day of week
            basic_string<Ch> str;
            while (ct.is(ctype_base::alpha,c)) {      // read characters into string
                str += c;
                if (++b == e) break;
                c = *b;
            }

            tm t;
            basic_stringstream<Ch> ss(str);
            typedef istreambuf_iterator<Ch> SI;          // iterator type for ss' buffer
            get_monthname(ss.rdbuf(),SI(),s,r,&t); // read from in-memory stream buffer

            if ((r&(ios_base::badbit|ios_base::failbit))==0) {
                *v= t.tm_mon;
                *res = month;
                return b;
            }

            r = 0;      // clear state before trying to read a second time
            get_weekday(ss.rdbuf(),SI(),s,r,&t);    // read from in-memory stream buffer

            if ((r&(ios_base::badbit|ios_base::failbit))==0) {
                *v = t.tm_wday;
                *res = dayofweek;
                return b;
            }
        }

        r |= ios_base::failbit;
        return b;
    }
```

The tricky part here is to distinguish months from weekdays. We read through input iterators, so we cannot read `[b,e)` twice, looking first for a month and then for a day. On the other hand, we cannot look at one character at a time and decide, because only ***get_monthname()*** and ***get_weekday()*** know which character sequences make up the names of the months and the names of the days of the week in a given locale. The solution I chose was to read strings of alphabetic characters into a ***string***, make a ***stringstream*** from that string, and then repeatedly read from that stream's ***streambuf***.

The error recording uses the state bits, such as ***ios_base::badbit***, directly. This is necessary because the more convenient functions for manipulating stream state, such as ***clear()*** and ***setstate()***, are defined in ***basic_ios*** rather than in its base ***ios_base*** (§21.3.3). If necessary, the >> operator then uses the error results reported by ***get_date()*** to reset the state of the input stream.

Given ***getval()***, we can read values first and then try to see whether they make sense later. The ***date_order()*** can be crucial:

```
template<class Ch, class In>
In Date_in<Ch,In>::do_get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* tmp) const
// optional day of week followed by ymd, dmy, mdy, or ydm
{
    int val[3];                     // for day, month, and year values in some order
    Vtype res[3] = { novalue };     // for value classifications

    for (int i=0; b!=e && i<3; ++i) {    // read day, month, and year
        b = getval(b,e,s,r,&val[i],&res[i]);
        if (r) return b;                 // oops: error
        if (res[i]==novalue) {           // couldn't complete date
            r |= ios_base::badbit;
            return b;
        }
        if (res[i]==dayofweek) {
            tmp->tm_wday = val[i];
            --i; // oops: not a day, month, or year
        }
    }

    time_base::dateorder order = date_order();      // now try to make sense of the values read

    if (res[0] == month) {              // mdy or error
        // ...
    }
    else if (res[1] == month) {         // dmy or ymd or error
        tmp->tm_mon = val[1];
        switch (order) {
        case dmy:
            tmp->tm_mday = val[0];
            tmp->tm_year = val[2];
            break;
        case ymd:
            tmp->tm_year = val[0];
            tmp->tm_mday = val[2];
            break;
        default:
            r |= ios_base::badbit;
            return b;
        }
    }
    else if (res[2] == month) {         // ydm or error
        // ...
    }
    else {                              // rely on dateorder or error
        // ...
    }

    tmp->tm_year -= 1900;    // adjust base year to suit tm convention
    return b;
}
```

I have omitted bits of code that do not add to the understanding of locales, dates, or the handling of input. Writing better and more general date input functions are left as exercises (§D.6[9-10]).

Here is a simple test program:

```
int main()
try {
        cin.imbue(loc(locale(),new Date_in));   // read Dates using Date_in

        while (cin >> dd && dd != Date()) cout << dd << endl;
}
catch (Date::Bad_date e) {
        cout << "bad date caught: " << e.why << endl;
}
```

Note that `do_get_date()` will accept meaningless dates, such as

```
Thursday October 7, 1998
```

and

```
1999/Feb/31
```

The checks for consistency of the year, month, day, and optional day of the week are done in *Date*'s constructor. It is the *Date* class' job to know what constitutes a correct date, and it is not necessary for *Date_in* to share that knowledge.

It would be possible to have `getval()` or `do_get_date()` guess about the meaning of numeric values. For example,

```
12 May 1922
```

is clearly not the day 1922 of year 12. That is, we could ''guess'' that a numeric value that couldn't be a day of the specified month must be a year. Such ''guessing'' can be useful in specific constrained context. However, it in not a good idea in more general contexts. For example,

```
12 May 15
```

could be a date in the year 12, 15, 1912, 1915, 2012, or 2015. Sometimes, a better approach is to augment the notation with clues that disambiguate years and days. For example, *1st* and *15th* are clearly days of a month. Similarly, *751BC* and *1453AD* are explicitly identified as years.

D.4.5 Character Classification

When reading characters from input, it is often necessary to classify them to make sense of what is being read. For example, to read a number, an input routine needs to know which letters are digits. Similarly, §6.1.2 showed a use of standard character classification functions for parsing input.

Naturally, classification of characters depends on the alphabet used. Consequently, a facet *ctype* is provided to represent character classification in a locale.

The character classes as described by an enumeration called *mask*:

```
class std::ctype_base {
public:
    enum mask {                 // the actual values are implementation defined
        space = 1,              // whitespace (in "C" locale: ' ', '\n', '\t', ...)
        print = 1<<1,           // printing characters
        cntrl = 1<<2,           // control characters
        upper = 1<<3,           // uppercase characters
        lower = 1<<4,           // lowercase characters
        alpha = 1<<5,           // alphabetic characters
        digit = 1<<6,           // decimal digits
        punct = 1<<7,           // punctuation characters
        xdigit = 1<<8,          // hexadecimal digits
        alnum=alpha|digit,      // alphanumeric characters
        graph=alnum|punct
    };
};
```

This ***mask*** doesn't depend on a particular character type. Consequently, this enumeration is placed in a (non-template) base class.

Clearly, ***mask*** reflects the traditional C and C++ classification (§20.4.1). However, for different character sets, different character values fall into different classes. For example, for the ASCII character set, the integer value *125* represents the character '}', which is a punctuation character (***punct***). However, in the Danish national character set, *125* represents the vowel 'å', which in a Danish locale must be classified as an ***alpha***.

The classification is called a ''mask'' because the traditional efficient implementation of character classification for small character sets is a table in which each entry holds bits representing the classification. For example:

```
table['a'] == lower|alpha|xdigit
table['1'] == digit
table[' '] == space
```

Given that implementation, ***table***[*c*]&*m* is nonzero if the character *c* is an *m* and *0* otherwise.

The ***ctype*** facet is defined like this:

```
template <class Ch>
class std::ctype : public locale::facet, public ctype_base {
public:
    typedef Ch char_type;
    explicit ctype(size_t r = 0);

    bool is(mask m, Ch c) const; // is "c" an "m"?

    // place classification for each Ch in [b:e) into v:
    const Ch* is(const Ch* b, const Ch* e, mask* v) const;

    const Ch* scan_is(mask m, const Ch* b, const Ch* e) const;   // find an m
    const Ch* scan_not(mask m, const Ch* b, const Ch* e) const; // find a non-m
```

```
        Ch toupper(Ch c) const;
        const Ch* toupper(Ch* b, const Ch* e) const; // convert [b:e)
        Ch tolower(Ch c) const;
        const Ch* tolower(Ch* b, const Ch* e) const;

        Ch widen(char c) const;
        const char* widen(const char* b, const char* e, Ch* b2) const;
        char narrow(Ch c, char def) const;
        const Ch* narrow(const Ch* b, const Ch* e, char def, char* b2) const;

        static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

    protected:
        ~ctype();

        // virtual ''do_'' functions for public functions (see §D.4.1)
    };
```

A call `is(m,c)` tests whether the character `c` belongs to the classification `m`. For example:

```
int count_spaces(const string& s, const locale& loc)
{
    const ctype<char>& ct = use_facet< ctype<char> >(loc);
    int i = 0;
    for(string::const_iterator p = s.begin(); p != s.end(); ++p)
        if (ct.is(ctype_base::space, *p)) ++i;      // whitespace as defined by ct
    return i;
}
```

Note that it is also possible to use `is()` to check whether a character belongs to one of a number of classifications. For example:

```
ct.is(ctype_base::space|ctype_base::punct,c); // is c whitespace or punctuation in ct?
```

A call `is(b,e,v)` determines the classification of each character in `[b,e)` and places it in the corresponding position in the array `v`.

A call `scan_is(m,b,e)` returns a pointer to the first character in `[b,e)` that is an `m`. If no character is classified as an `m`, `e` is returned. As ever for standard facets, the public member function is implemented by a call to its ''`do_`'' virtual function. A simple implementation might be:

```
template <class Ch>
const Ch* std::ctype<Ch>::do_scan_is(mask m, const Ch* b, const Ch* e) const
{
    while (b!=e && !is(m,*b)) ++b;
    return b;
}
```

A call `scan_not(m,b,e)` returns a pointer to the first character in `[b,e)` that is not an `m`. If all characters are classified as `m`, `e` is returned.

A call `toupper(c)` returns the uppercase version of `c` if such a version exists in the character set used and `c` itself otherwise.

A call `toupper(b,e)` converts each character in the range `[b,e)` to uppercase and returns `e`. A simple implementation might be:

```
template <class Ch>
const Ch* std::ctype<Ch>::to_upper(Ch* b, const Ch* e)
{
    for (; b!=e; ++b) *b = toupper(*b);
    return e;
}
```

The *tolower()* functions are similar to *toupper()* except that they convert to lowercase.

A call *widen(c)* transforms the character *c* into its corresponding *Ch* value. If *Ch*'s character set provides several characters corresponding to *c*, the standard specifies that ''the simplest reasonable transformation'' be used. For example,

```
wcout << use_facet< ctype<wchar_t> >(wcout.getloc()).widen('e');
```

will output a reasonable equivalent to the character *e* in *wcout*'s locale.

Translation between unrelated character representations, such as ASCII and EBCDIC, can also be done by using *widen()*. For example, assume that an *ebcdic* locale exists:

```
char EBCDIC_e = use_facet< ctype<char> >(ebcdic).widen('e');
```

A call *widen(b,e,v)* takes each character in the range *[b,e)* and places a widened version in the corresponding position in the array *v*.

A call *narrow(ch,def)* produces a *char* value corresponding to the character *ch* from the *Ch* type. Again, ''the simplest reasonable transformation'' is to be used. If no such corresponding *char* exist, *def* is returned.

A call *narrow(b,e,def,v)* takes each character in the range *[b,e)* and places a narrowed version in the corresponding position in the array *v*.

The general idea is that *narrow()* converts from a larger character set to a smaller one and that *widen()* performs the inverse operation. For a character *c* from the smaller character set, we expect:

```
c == narrow(widen(c),0) // not guaranteed
```

This is true provided that the character represented by *c* has only one representation in ''the smaller character set.'' However, that is not guaranteed. If the characters represented by a *char* are not a subset of those represented by the larger character set (*Ch*), we should expect anomalies and potential problems with code treating characters generically.

Similarly, for a character *ch* from the larger character set, we might expect:

```
widen(narrow(ch,def)) == ch || widen(narrow(ch,def)) == widen(def)  // not guaranteed
```

However, even though this is often the case, it cannot be guaranteed for a character that is represented by several values in the larger character set but only once in the smaller character set. For example, a digit, such as *7*, often has several separate representations in a large character set. The reason for that is typically that a large character set has several conventional character sets as subsets and that the characters from the smaller sets are replicated for ease of conversion.

For every character in the basic source character set (§C.3.3), it is guaranteed that

```
widen(narrow(ch_lit,0)) == ch_lit
```

For example:

```
widen(narrow('x',0)) == 'x'
```

The *narrow()* and *widen()* functions respect character classifications wherever possible. For example, if *is(alpha,c)*, then *is(alpha,narrow(c,'a'))* and *is(alpha,widen(c))* wherever *alpha* is a valid mask for the locale used.

A major reason for using a *ctype* facet in general and for using *narrow()* and *widen()* functions in particular is to be able to write code that does I/O and string manipulation for any character set; that is, to make such code generic with respect to character sets. This implies that *iostream* implementations depend critically on these facilities. By relying on *<iostream>* and *<string>*, a user can avoid most direct uses of the *ctype* facet.

A *_byname* version (§D.4, §D.4.1) of *ctype* is provided:

```
template<class Ch> class std::ctype_byname : public ctype<Ch> { /* ... */ };
```

D.4.5.1 Convenience Interfaces

The most common use of the *ctype* facet is to inquire whether a character belongs to a given classification. Consequently, a set of functions is provided for that:

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

These functions are trivially implemented by using *use_facet*. For example:

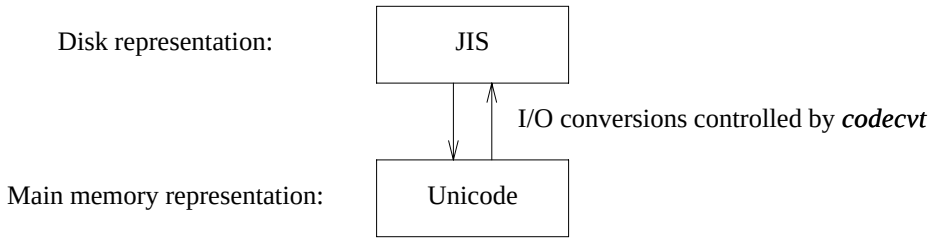
```
template<class Ch>
inline bool isspace(Ch c, const locale& loc)
{
    return use_facet< ctype<Ch> >(loc).is(space,c);
}
```

The one-argument versions of these functions, presented in §20.4.2, are simply these functions for the current C global locale (not the global C++ locale, *locale()*). Except for the rare cases in which the C global locale and the C++ global locale differ (§D.2.3), we can think of a one-argument version as the two-argument version applied to *locale()*. For example:

```
inline int isspace(int i)
{
    return isspace(i,locale());    // almost
}
```

D.4.6 Character Code Conversion

Sometimes, the representation of characters stored in a file differs from the desired representation of those same characters in main memory. For example, Japanese characters are often stored in files in which indicators (''shifts'') tell to which of the four common character sets (kanji, katakana, hiragana, and romaji) a given sequence of characters belongs. This is a bit unwieldy because the meaning of each byte depends on its ''shift state,'' but it can save memory because only a kanji requires more than one byte for its representation. In main memory, these characters are easier to manipulate when represented in a multi-byte character set where every character has the same size. Such characters (for example, Unicode characters) are typically placed in wide characters (***wchar_t***; §4.3). Consequently, the ***codecvt*** facet provides a mechanism for converting characters from one representation to another as they are read or written. For example:



This code-conversion mechanism is general enough to provide arbitrary conversions of character representations. It allows us to write a program to use a suitable internal character representation (stored in ***char***, ***wchar_t***, or whatever) and to then accept a variety of input character stream representations by adjusting the locale used by iostreams. The alternative would be to modify the program itself or to convert input and output files from/to a variety of formats.

The ***codecvt*** facet provides conversion between different character sets when a character is moved between a stream buffer and external storage:

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

template <class I, class E, class State>
class std::codecvt : public locale::facet, public codecvt_base {
public:
    typedef I intern_type;
    typedef E extern_type;
    typedef State state_type;

    explicit codecvt(size_t r = 0);

    result in(State&, const E* from, const E* from_end, const E*& from_next, // read
                  I* to, I* to_end, I*& to_next) const;
```

```
        result out(State&, const I* from, const I* from_end, const I*& from_next, // write
                   E* to, E* to_end, E*& to_next) const;

        result unshift(State&, E* to, E* to_end, E*& to_next) const; // end character sequence

        int encoding() const throw();              // characterize basic encoding properties
        bool always_noconv() const throw();        // can we do I/O without code translation?

        int length(const State&, const E* from, const E* from_end, size_t max) const;
        int max_length() const throw();            // maximum possible length()

        static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

    protected:
        ~codecvt();

        // virtual ''do_'' functions for public functions (see §D.4.1)
    };
```

A ***codecvt*** facet is used by ***basic_filebuf*** (§21.5) to read or write characters. A ***basic_filebuf*** obtains this facet from the stream's locale (§21.7.1).

The ***State*** template argument is the type used to hold the shift state of the stream being converted. ***State*** can also be used to identify different conversions by specifying a specialization. The latter is useful because characters of a variety of character encodings (character sets) can be stored in objects of the same type. For example:

```
class JISstate { /* .. */ };

p = new codecvt<wchar_t, char, mbstate_t>;     // standard char to wide char
q = new codecvt<wchar_t, char, JISstate>;      // JIS to wide char
```

Without the different ***State*** arguments, there would be no way for the facet to know which encoding to assume for the stream of ***char***s. The ***mbstate_t*** type from <***cwchar***> or <***wchar.h***> identifies the system's standard conversion between ***char*** and ***wchar_t***.

A new ***codecvt*** can be also created as a derived class and identified by name. For example:

```
class JIScvt : public codecvt<wchar_t, char, mbstate_t> { /* ... */ };
```

A call ***in***(***s***, ***from***, ***from_end***, ***from_next***, ***to***, ***to_end***, ***to_next***) reads each character in the range [***from***, ***from_end***) and tries to convert it. If a character is converted, ***in***() writes its converted form to the corresponding position in the [***to***, ***to_end***) range; if not, ***in***() stops at that point. Upon return, ***in***() stores the position one-beyond-the-last character read in ***from_next*** and the position one-beyond-the-last character written in ***to_next***. The ***result*** value returned by ***in***() indicates how much work was done:

ok:	all characters in the [***from***, ***from_end***) range converted
partial:	not all characters in the [***from***, ***from_end***) range were converted
error:	***in***() encountered a character it couldn't convert
noconv:	no conversion was needed

Note that a ***partial*** conversion is not necessarily an error. Possibly more characters have to be read before a multibyte character is complete and can be written, or maybe the output buffer has to be emptied to make room for more characters.

The *s* argument of type *State* indicates the state of the input character sequence at the start of the call of *in*(). This is significant when the external character representation uses shift states. Note that *s* is a (non-*const*) reference argument: At the end of the call, *s* holds the state of shift state of the input sequence. This allows a programmer to deal with *partial* conversions and to convert a long sequence using several calls to *in*().

A call *out*(*s*,*from*,*from_end*,*from_next*,*to*,*to_end*,*to_next*) converts [*from*,*from_end*) from the internal to the external representation in the same way the *in*() converts from the external to the internal representation.

A character stream must start and end in a ''neutral'' (unshifted) state. Typically, that state is *State*(). A call *unshift*(*s*,*to*,*to_end*,*to_next*) looks at *s* and places characters in [*to*,*to_end*) as needed to bring a sequence of characters back to that unshifted state. The result of *unshift*() and the use of *to_next* are done just like *out*().

A call *length*(*s*,*from*,*from_end*,*max*) returns the number of characters that *in*() could convert from [*from*,*from_end*).

A call *encoding*() returns

- *–1* if the encoding of the external character set uses state (for example, uses shift and unshift character sequences)
- *0* if the encoding uses varying number of bytes to represent individual characters (for example, a character representation might use a bit in a byte to indicate whether one or two bytes are used to represents that character)
- *n* if every character of the external character representation is *n* bytes

A call *always_noconv*() returns *true* if no conversion is required between the internal and the external character sets and *false* otherwise. Clearly, *always_noconv*()==*true* opens the possibility for the implementation to provide the maximally efficient implementation that simply doesn't invoke the conversion functions.

A call *max_length*() returns the maximum value that *length*() can return for a valid set of arguments.

The simplest code conversion that I can think of is one that converts input to uppercase. Thus, this is about as simple as a *codecvt* can be and still perform a service:

```
class Cvt_to_upper : public codecvt<char,char,mbstate_t> {     // convert to uppercase
    explicit Cvt_to_upper(size_t r = 0) : codecvt(r) { }
protected:
    // read external representation write internal representation:
    result do_in(State& s, const char* from, const char* from_end, const char*& from_next,
                 char* to, char* to_end, char*& to_next) const;

    // read internal representation write external representation:
    result do_out(State& s, const char* from, const char* from_end, const char*& from_next,
                  char* to, char* to_end, char*& to_next) const
    {
        return codecvt<char,char,mbstate_t>::do_out
            (s,from,from_end,from_next,to,to_end,to_next);
    }
```

```
        result do_unshift(State&, E* to, E* to_end, E*& to_next) const { return ok; }

        int do_encoding() const throw() { return 1; }
        bool do_always_noconv() const throw() { return false; }

        int do_length(const State&, const E* from, const E* from_end, size_t max) const;
        int do_max_length() const throw();      // maximum possible length()
};

codecvt<char,char,mbstate_t>::result
Cvt_to_upper::do_in(State& s, const char* from, const char* from_end,
                    const char*& from_next, char* to, char* to_end, char*& to_next) const
{
        // ... §D.6[16] ...
}

int main()      // trivial test
{
        locale ulocale(locale(), new Cvt_to_upper);

        cin.imbue(ulocale);

        char ch;
        while (cin>>ch) cout << ch;
}
```

A *_byname* version (§D.4, §D.4.1) of *codecvt* is provided:

```
template <class I, class E, class State>
class std::codecvt_byname : public codecvt<I,E,State> { /* ... */ };
```

D.4.7 Messages

Naturally, most end users prefer to use their native language to interact with a program. However, we cannot provide a standard mechanism for expressing *locale*-specific general interactions. Instead, the library provides a simple mechanism for keeping a *locale*-specific set of strings from which a programmer can compose simple messages. In essence, *messages* implements a trivial read-only database:

```
class std::messages_base {
public:
        typedef int catalog;  // catalog identifier type
};

template <class Ch>
class std::messages : public locale::facet, public messages_base {
public:
        typedef Ch char_type;
        typedef basic_string<Ch> string_type;

        explicit messages(size_t r = 0);
```

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

        static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)

    protected:
        ~messages();

        // virtual ''do_'' functions for public functions  (see §D.4.1)
    };
```

A call *open*(*s*,*loc*) opens a ''catalog'' of messages called *s* for the locale *loc*. A catalog is a set of strings organized in an implementation-specific way and accessed through the *messages*::*get*() function. A negative value is returned if no catalog named *s* can be opened. A catalog must be opened before the first use of *get*().

A call *close*(*cat*) closes the catalog identified by *cat* and frees all resources associated with that catalog.

A call *get*(*cat*,*set*,*id*,"*foo*") looks for a message identified by (*set*,*id*) in the catalog *cat*. If a string is found, *get*() returns that string; otherwise, *get*() returns the default string (here, *string*("*foo*")).

Here is an example of a *messages* facet for an implementation in which a message catalog is a vector of sets of ''messages'' and a ''message'' is a string:

```
    struct Set {
        vector<string> msgs;
    };

    struct Cat {
        vector<Set> sets;
    };

    class My_messages : public messages<char> {
        vector<Cat>& catalogs;
    public:
        explicit My_messages(size_t = 0) :catalogs(*new vector<Cat>) { }

        catalog do_open(const string& s, const locale& loc) const;      // open catalog s
        string do_get(catalog c, int s, int m, const string&) const;    // get message (s,m) in c
        void do_close(catalog cat) const
        {
            if (catalogs.size()<=cat) catalogs.erase(catalogs.begin()+cat);
        }

        ~My_messages() { delete &catalogs; }
    };
```

All *messages*' member functions are *const*, so the catalog data structure (the *vector*<*Set*>) is stored outside the facet.

A message is selected by specifying a catalog, a set within that catalog, and a message string within that set. A string is supplied as an argument, to be used as a default result in case no message is found in the catalog:

```
string My_messages::do_get(catalog cat, int set, int msg, const string& def) const
{
    if (catalogs.size()<=cat) return def;
    Cat& c = catalogs[cat];
    if (c.sets.size()<=set) return def;
    Set& s = c.sets[set];
    if (s.msgs.size()<=msg) return def;
    return s.msgs[msg];
}
```

Opening a catalog involves reading a textual representation from disk into a *Cat* structure. Here, I chose a representation that is trivial to read. A set is delimited by <<< and >>>, and each message is a line of text:

```
messages<char>::catalog My_messages::do_open(const string& n, const locale& loc) const
{
    string nn = n + locale().name();
    ifstream f(nn.c_str());
    if (!f) return -1;

    catalogs.push_back(Cat());          // make in-core catalog
    Cat& c = catalogs.back();
    string s;
    while (f>>s && s=="<<<") {          // read Set
        c.sets.push_back(Set());
        Set& ss = c.sets.back();
        while (getline(f,s) && s != ">>>") ss.msgs.push_back(s);     // read message
    }
    return catalogs.size()-1;
}
```

Here is a trivial use:

```
int main()
{
    if (!has_facet< My_messages >(locale())) {
        cerr << "no messages facet found in " << locale().name() << '\n';
        exit(1);
    }

    const messages<char>& m = use_facet< My_messages >(locale());
    extern string message_directory;    // where I keep my messages
    int cat = m.open(message_directory, locale());
    if (cat<0) {
        cerr << "no catalog found\n";
        exit(1);
    }

    cout << m.get(cat,0,0,"Missed again!") << endl;
    cout << m.get(cat,1,2,"Missed again!") << endl;
```

```
        cout << m.get(cat,1,3,"Missed again!") << endl;
        cout << m.get(cat,3,0,"Missed again!") << endl;
}
```

If the catalog is

```
<<<
hello
goodbye
>>>
<<<
yes
no
maybe
>>>
```

this program prints

```
hello
maybe
Missed again!
Missed again!
```

D.4.7.1 Using Messages from Other Facets

In addition to being a repository for ***locale***-dependent strings used to communicate with users, messages can be used to hold strings for other facets. For example, the ***Season_io*** facet (§D.3.2) could have been written like this:

```
class Season_io : public locale::facet {
        const messages<char>& m;        // message directory
        int cat;                        // message catalog
public:
        class Missing_messages { };

        Season_io(int i = 0)
                : locale::facet(i),
                  m(use_facet<Season_messages>(locale())),
                  cat(m.open(message_directory,locale()))
        { if (cat<0) throw Missing_messages(); }

        ~Season_io() { }      // to make it possible to destroy Season_io objects (§D.3)

        const string& to_str(Season x) const;              // string representation of x

        bool from_str(const string& s, Season& x) const;   // place Season corresponding to s in x

        static locale::id id; // facet identifier object (§D.2, §D.3, §D.3.1)
};

locale::id Season_io::id; // define the identifier object
```

```
const string& Season_io::to_str(Season x) const
{
    return m->get(cat,x,"no-such-season");
}

bool Season_io::from_str(const string& s, Season& x) const
{
    for (int i = Season::spring; i<=Season::winter; i++)
        if (m->get(cat,i,"no-such-season") == s) {
            x = Season(i);
            return true;
        }
    return false;
}
```

This ***messages***-based solution differs from the original solution (§D.3.2) in that the implementer of a set of ***Season*** strings for a new locale needs to be able to add them to a ***messages*** directory. This is easy for someone adding a new locale to an execution environment. However, since ***messages*** provides only a read-only interface, adding a new set of season names may be beyond the scope of an application programmer.

A ***_byname*** version (§D.4, §D.4.1) of ***messages*** is provided:

```
template <class Ch>
class std::messages_byname : public messages<Ch> { /* ... */ };
```

D.5 Advice

[1] Expect that every nontrivial program or system that interacts directly with people will be used in several different countries; §D.1.
[2] Don't assume that everyone uses the same character set as you do; §D.4.1.
[3] Prefer using ***locale***s to writing ad hoc code for culture-sensitive I/O; §D.1.
[4] Avoid embedding locale name strings in program text; §D.2.1.
[5] Minimize the use of global format information; §D.2.3, §D.4.4.7.
[6] Prefer locale-sensitive string comparisons and sorts; §D.2.4, §D.4.1.
[7] Make ***facet***s immutable; §D.2.2, §D.3.
[8] Keep changes of ***locale*** to a few places in a program; §D.2.3.
[9] Let ***locale*** handle the lifetime of ***facet***s; §D.3.
[10] When writing locale-sensitive I/O functions, remember to handle exceptions from user-supplied (overriding) functions; §D.4.2.2.
[11] Use a simple ***Money*** type to hold monetary values; §D.4.3.
[12] Use simple user-defined types to hold values that require locale-sensitive I/O (rather than casting to and from values of built-in types); §D.4.3.
[13] Don't believe timing figures until you have a good idea of all factors involved; §D.4.4.1.
[14] Be aware of the limitations of ***time_t***; §D.4.4.1, §D.4.4.5.
[15] Use a date-input routine that accepts a range of input formats; §D.4.4.5.
[16] Prefer the character classification functions in which the locale is explicit; §D.4.5, §D.4.5.1.

D.6 Exercises

1. (∗2.5) Define a ***Season_io*** (§D.3.2) for a language other than American English.
2. (∗2) Define a ***Season_io*** (§D.3.2) class that takes a set of name strings as a constructor argument so that ***Season*** names for different locales can be represented as objects of this class.
3. (∗3) Write a ***collate<char>::compare()*** that gives dictionary order. Preferably, do this for a language, such as German or French, that has more letters in its alphabet than English does.
4. (∗2) Write a program that reads and writes ***bool***s as numbers, as English words, and as words in another language of your choice.
5. (∗2.5) Define a ***Time*** type for representing time of day. Define a ***Date_and_time*** type by using ***Time*** and a ***Date*** type. Discuss the pros and cons of this approach compared to the ***Date*** from (§D.4.4). Implement ***locale***-sensitive I/O for ***Time*** and ***Date_and_time***.
6. (∗2.5) Design and implement a postal code (zip code) facet. Implement it for at least two countries with dissimilar conventions for writing addresses. For example: ***NJ 07932*** and ***CB2***1QA .
7. (∗2.5) Design and implement a phone number facet. Implement it for at least two countries with dissimilar conventions for writing phone numbers. For example, ***(973) 360-8000*** and ***1223 343000***.
8. (∗2.5) Experiment to find out what input and output formats your implementation uses for date information.
9. (∗2.5) Define a ***get_time()*** that ''guesses'' about the meaning of ambiguous dates, such as 12/5/1995, but still rejects all or almost all mistakes. Be precise about what ''guesses'' are accepted, and discuss the likelihood of a mistake.
10. (∗2) Define a ***get_time()*** that accepts a greater variety of input formats than the one in §D.4.4.5.
11. (∗2) Make a list of the locales supported on your system.
12. (∗2.5) Figure out where named locales are stored on your system. If you have access to the part of the system where locales are stored, make a new named locale. Be very careful not to break existing locales.
13. (∗2) Compare the two ***Season_io*** implementations (§D.3.2 and §D.4.7.1).
14. (∗2) Write and test a ***Date_out*** facet that writes ***Date***s using a format supplied as a constructor argument. Discuss the pros and cons of this approach compared to the global date format provided by ***date_fmt*** (§D.4.4.6).
15. (∗2.5) Implement I/O of Roman numerals (such as ***XI*** and ***MDCLII***).
16. (∗2.5) Implement and test ***Cvt_to_upper*** (§D.4.6).
17. (∗2.5) Use ***clock()*** to determine average cost of (1) a function call, (2) a virtual function call, (3) reading a ***char***, (4) reading a 1-digit ***int***, (5) reading a 5-digit ***int***, (6) reading a 5-digit ***double***, (7) a 1-character ***string***, (8) a 5-character ***string***, and (9) a 40-character ***string***.
18. (∗6.5) Learn another natural language.