Platform-Independent Code Conversion within the C++ Locale Framework

Lars Engebretsen

Department of Numerical Analysis and Computer Science,
Royal Institute of Technology,
SE-100 44 Stockholm
SWEDEN

February, 2005

Abstract. This paper describes some of the author's experiences from a C++ implementation of a concordance program for texts in Old West Norse (also known as Old Icelandic) and Runic Swedish. Since the input to the program used a character repertoire that no standard one-byte character encoding covers, it was natural to use Unicode to represent data both inside the program and in external files. Inside the program, each character was represented with C++ "wide characters"; the input and output was represented in UTF-8. The author constructed C++ code conversion facets that convert data between those two representations during file I/O. This enabled him to successfully compile, and run, the concordance program on both Linux (Fedora Core 3 with gcc 3.4.2) and Windows XP (using Visual C++ .NET 2003). The only necessary change to the source when changing platform was isolated to the lines selecting which code conversion facet to use—all other pieces of code remained unchanged. In particular, the author could still use the standard C++ locale framework for collation and code conversion, in spite of the fact that the library-provided code conversion facets had been replaced.

Key words. C++, Locale, Code Conversion

1 INTRODUCTION

Between 1993 and 1997, a database covering all Scandinavian runic inscriptions was compiled [1, 5]. By the time of this writing (February 2005), the latest update of the database, announced in September 2004, is available on URL «http://www.nordiska.uu.se/samnord.html». This paper documents the experiences drawn from the implementation of a concordance program for the entire database. While the task of generating a concordance is well understood—and in this case particularly easy since the entire database and the concordance together fit in memory on any modern desktop computer—the fact that the database contains texts in arcane languages puts focus on some technicalities in the implementation. The characters required to properly represent the text in the database are not contained in any standard one-byte character encoding. Therefore, the author decided to store the text in the database using Unicode

encoded with UTF-8. Inside the concordance program, however, it would be easier to work with an encoding where each character is represented by the same number of bytes. Also, the sorting keys used to construct the concordance had to be converted into lower case by the concordance program. Finally, it was desirable, although not strictly necessary, to collate the resulting concordance according to slightly non-standard collation rules.

The C++ locale framework aims at giving a standardized interface to concepts that are inherently system dependent, namely how software is adapted to users' language and cultural conventions. There are several components in a locale, such as formatting of numbers, formatting of dates, formatting of monetary values, collation of strings, classification of characters, and code conversion—in C++ these components are modeled by so called *facets*. The by far most portable way to perform case conversion and collation of arbitrary text is to use the corresponding facets of the standard locale provided by the system. There are standard C++ library functions that operate on C++ strings, i.e., on objects of type `std::string` and `std::wstring`. For the application considered in this paper, the natural choice was to use wide strings inside the program and UTF-8 encoded text files for input and output. To read from a file stream corresponding to a file where the characters are encoded with UTF-8 into into C++ wide strings requires code conversion. Naturally, code conversion is also required when writing to UTF-8 encoded files. For this task, the standard C++ locale framework contains so called *code conversion facets*. However, while most parts of the C++ locale framework are well described, not only in the C++ standard [3] but also in books [4, 6], the aspect of code conversion is only described to any sufficient detail in the C++ standard [3]. The reason for this is no doubt partly that the interaction between code conversion facets and I/O streams is the most implementation specific parts of the locale framework. However, it also seems that the code conversion facets were introduced into the standard at a relatively late stage. In fact, previous editions of the C++ standard [3] had slight ambiguities in the definition of the code conversion parts of the locale framework. Hence, any practical experiences from code conversion using the standard C++ locale framework are of general interest.

The purpose of this paper is to describe such experiences, obtained by the author through an implementation of the concordance program. In particular, the program was designed with the intention that it should be possible to compile, and run, it on as many platforms as possible, also on platforms lacking UTF-8 support in their standard locales. This platform-independence should ideally come at a low cost, i.e., the amount of author-written code dealing with the details of code conversion, case conversion and collation should be small. Naturally, this introduces a trade-off. True platform-independence could be achieved by completely removing the dependence on the locale framework provided by standard C++ and instead writing own routines. Using, on the other hand, only functions provided by the standard C++ library, the amount of extra code is reduced to a minimum but then the program is restricted to the func-

tionality provided by the surrounding operating system. The author settled for a compromise between those two extremes. After a short overview of the runic-text database and the concordance program, this compromise is described, and motivated, below.

2 THE DATABASE AND THE CONCORDANCE PROGRAM

Before delving into the aspects of code conversion, case conversion and collation, we briefly describe the rest of the concordance program and the runic-text database.

The aim of the Scandinavian Runic-text database project, conducted in 1993–1997 [1], was to collect all Scandinavian runic texts in a single corpus. The database consists of several text files. One file contains a transliteration of the inscription, i.e., every grapheme (rune) is represented by a unique character. Within the field of runic studies, there are well-established conventions for how runes should be transliterated. The version of the database available for download from «http://www.nordiska.uu.se/samnord.html» does not quite follow those conventions, though. In her presentation of the project, Lena Peterson wrote that "the limitations of the computer keyboard forces us to make use of some unconventional signs" [5, p 307]. With the advent of Unicode [7] it is no longer necessary to use hacks and work-arounds; the four non-ASCII characters needed to transliterate runes are now readily available: 1) "Latin small letter ae" æ (U+00E6). 2) "Latin small letter o with stroke" ø (U+00F8), 3) "Latin small letter eng": ŋ (U+014B). 4) "Latin letter small capital r": ʀ (U+0280).

Another file in the database contains word-by-word translations of the inscriptions into normalized Old West Norse, also known as Old Icelandic. Since the spelling of the same word varies greatly from inscription to inscription, this file enables researchers to search for all occurrences of a particular word. Old West Norse is conventionally written with quite a few non-ASCII characters; see for instance Table 1 which shows parts of some inscriptions containing the word "runes". In addition to the accented vowels and the letters æ, Æ, ø, Ø, þ, Þ, and ð, which are all covered by the Latin-1 character set [2], the following characters are needed: "Latin ligature oe" – Œ (U+0152) and œ (U+0153) – together with "Latin letter o with ogonek" – Ǫ (U+01EA) and ǫ (U+01EB).

Each of the files in the runic-text database project contains one inscription per line, hence the database is read by the concordance program on a line-by-line basis. As mentioned above, the author converted the database files to UTF-8 and used the "proper" characters for transliterations of inscriptions and translations of inscriptions into Old West Norse. The conversion between UTF-8 and C++ wide characters during file I/O is described below.

Inside the concordance program, each line in Table 1 corresponds to a pair of a C++ `std::wstring` (the key) and a C++ object of type `Entry` (the line and the index of the word in the line). The inner workings of this latter object is omitted

	rúnar
DR 209	sinn ok hans kona ept ver sinn. En Sóti reist rúnar þessar ept dróttin sinn. Þórr vígi þessar
DR 209	rúnar þessar ept dróttin sinn. Þórr vígi þessar rúnar. §C At ræta(?) sá verði er stein þenna
G 203	bjartr á bergi, en bró fyrir. §D Hróðbjǫrn risti rúnar þessar, Geirl[e]ifr sumar, er gørla kann.
HS 11	Guð hjalpi sálu hans. Þeir mǫrkuðu rúnar, Ǫlvir ok Brandr.
HS 14	En þá Guðrún. Freymundr Fégylfa sonr fáði rúnar þessar. Vér sóttum stein þenna norðr í
N 393	Dróttinn hjalpi þeim manni, er þessar rúnar reist, svá þeim, er þær ræðr.
ÖG 64	stein þenna ept Greip, gilda sinn, Lófi reist rúnar þessar, Judda/Júta son.
ÖG 136	Eptir Vémóð/Vámóð standa rúnar þær. En Varinn fáði, faðir, eptir feigjan
SÖ 54	landbornir menn, létu rétta stein. Steinkell reist rúnar.
SÖ 333	í Kalmarna sundum, fóru af Skáney. Áskell risti rúnar þessar.
SÖ 205	at bróður sinn. Ásbjǫrn ok Tíðkumi hjoggu rúnar. Órœkja(?) steindi(?).
U 532	sinn. Guð hjalpi ǫnd hans. Þorbjǫrn Skald hjó rúnar.
U 654	drepinn. Guð hjalpi ǫnd þeira. Alríkr(?) reist-ek rúnar. Er kunni vel knerri stýra.
U 687	var dauðr í Holmgarði í Ólafs kirkju. Œpir risti rúnar.
U 897	Sigviðr, sonr Gillaugar, reisti rúnar eptir Ragnelfi, sværu sína.

Table 1. Some of the 300 entries corresponding to the word *rúnar* (runes) in the concordance. The left column shows the index of the inscription. This table contains lower case versions of all characters needed to write normalized Old Norse.

since it is of no interest for the main topic of this paper. For every word on every line, the corresponding key and `Entry` object is constructed and stored in a C++ multimap. As can be seen from Table 1, some words need to be "cleaned" before they can become proper keys. Specifically, punctuation characters, parentheses and braces must be removed. This is easy to accomplish with standard C++ string functions once the word has been stored in a `std::wstring`. The keys are also converted to lower case—unless they are actually the name of a person or a place—using the C++ locale framework as described below.

Once the entire database file has been processed as outlined above, the resulting concordance is output by iterating through the multimap. In fact, it turns out that by using the C++ locale framework it is possible to get the multimap containing the concordance sorted as new elements are inserted. The rest of this paper describes in detail the main interesting, from a modern programmer's point of view, parts of the concordance program: the conversion between UTF-8 and the internal character representation during file I/O, the case conversion, and the collation.

3 CODE CONVERSION

In the C++ standard library, *code conversion* takes place when reading from, and writing to, buffered streams. The problem addressed by code conversion is that when accessing a stream corresponding to some file, it may happen that the characters in the file are encoded in some way that is specific to the cultural conventions of the user of the program. In particular, the *user's locale* usually specifies the user's default character encoding. The simplest possible conversion is, of course, to not convert at all, and this is also usually the default conversion

applied to normal, "narrow" streams. For wide character streams, however, the situation is more complex. Let us remark, that "wide" in this context refers to the way the stream is seen from within the C++ programs. Reading from a wide C++ stream produces wide C++ characters, i.e., characters of type `wchar_t`, no matter what the underlying file looks like. Roughly speaking, the bytes read one by one from, say, a file must therefore be converted into Unicode characters; note that it may happen that several bytes correspond to one Unicode character.

To be more precise, the C++ standard mandates that the C++ type `wchar_t` should be able to "represent distinct codes for all members of the largest extended character set specified among the supported locales" [3, § 3.9.1, ¶ 5]. Hence, the standard does not require implementations to use Unicode as the encoding of wide characters, even though this is typically the case. Note, also, that the *size* of type `wchar_t` is not specified by the standard—on Unix systems it is typically 32 bits while it is 16 bits in, e.g., Windows XP. A consequence of this is that code conversion facets are inherently platform dependent, at least if they convert to and from the C++ type `wchar_t`. Using the type `wchar_t` is a natural choice since that type is supported by the standard C++ locale framework; the routines for collation and case conversion provided by the C++ standard library can then be used without any problems. The only other datatype supported by the locale framework in the C++ standard library is `char`, but, as mentioned above, no character encoding that covers all characters needed in the concordance would fit in a `char`.

3.1 Using library-provided code conversion facets

From the discussion above, it would seem that the easiest possible solution—from the programmer's point of view—is to rely on the code conversion facets provided by the C++ runtime libraries. This is, however, insufficient for the purposes of the concordance program—at least given the convention that the database files are fixed and use UTF-8 encoding or, indeed, any other fixed encoding. The main problem is that the code conversion facets provided by the C++ environment typically use the operating system's locale framework to decide what kind of code conversion to perform. On Unix systems, it is generally possible to change only the code conversion aspects of the locale by modifying the environment variable `LC_CTYPE`. In, e.g., Windows XP it seems that the locale is more monolithic: If, for instance, Swedish or Icelandic collation is desired, it appears to be impossible to specify UTF-8 as the character encoding using the standard operating system locale framework. The C++ locale framework is far more flexible than this since individual facets may be changed in a locale, thus producing a new locale. Alas, letting the operating system's locale specify code conversion makes it impossible to take advantage of this flexibility in a Windows XP environment. Therefore, there is only one way to get reasonable platform-independence and at the same time the flexibility of specifying that data files are always in UTF-8 no matter the locale selected by the user:

The application programmer must specify the code conversion explicitly, using an explicitly specified code conversion facet.

3.2 Implementing and using one's own C++ code conversion facet

A code conversion facet provides means for converting between *external* and *internal* representations of characters. In our case, the external representation is a sequence of bytes that should be interpreted as characters encoded according to UTF-8; the internal representation is a sequence of `wchar_t`. On a slightly more detailed level, these conversion capabilities are implemented by functions that convert some sequence from one representation to the other. To slightly complicate things, it may happen that several characters of one representation are needed to produce one character in the other representation, and vice versa. The conversion could also be *stateful*, i.e., there could be some kind of shift sequence stating that all subsequent characters should be given special treatment until an unshift sequence occurs. A detailed description of C++ code conversion facets can be found in the C++ standard [3, § 22.2.1.5].

As an illustration, the code conversion facet converting from UTF-8 to UTF-32, i.e., to Unicode characters represented as 32-bit wide characters, is included in Appendix A. The functions `do_in()` and `do_out()` perform the actual conversion; there is also a function `do_length()` that computes the number of wide characters in the internal representation needed to represent some given sequence of characters in the external representation. The functions `do_in()` and `do_out()` also check for *ill-formed code units*, see the Unicode standard [7, § 3.9] for the precise details regarding ill-formed code units in the different Unicode encoding forms.

Since `wchar_t` only holds 16 bits in Windows XP, the code conversion facet given in Appendix A would not work in that environment. Instead, a slightly modified facet must be used, that produces UTF-16 code units, with so called "surrogate pairs" as needed, in the internal representation. The resulting code conversion facet is very similar to the one in Appendix A.

Once the new code conversion facet has been specified it is straightforward to create a locale that uses it: the C++ class `std::locale` has a constructor that constructs a new locale object from an old one with a new facet.

```
std::codecvt<wchar_t, char, mbstate_t> *fromUTF8;
if(runningWindows)
  fromUTF8 = new UTF8_UTF16<wchar_t, char>(); // 16-bit wchar_t
else
  fromUTF8 = new UTF8_UTF32<wchar_t, char>(); // 32-bit wchar_t
// Use the default locale with the UTF8 code conversion facet added.
std::locale loc(std::locale(""), fromUTF8);
```

It is not enough to just create a new locale for the new code conversion to apply; In order to actually use the facet one imbues the desired files with the locale.

```
std::wifstream in("datafile.txt");
in.imbue(loc);
```

After that, the specified code conversion will be applied by the C++ standard library functions during file I/O. It is, consequently, possible to use some code conversion facet for reading some specified file, another facet for reading from some other specified file and a third facet for writing output, should such behaviour be desired.

Note that the choice of code conversion facet, although platform dependent, need only be present at one place in the source code. The choice can either be dynamic, as above, or static, governed by some configuration script or preprocessor directive. Hence, we do not obtain true platform independence, but we have isolated the platform dependence to a couple of lines at one place in the C++ source code. This slight inconvenience gives us the freedom of combining any collation order—specified in the "usual" way by the user selecting a certain locale according to the conventions of his or her particular platform—with file I/O using UTF-8 encoded files. For this project the author modified the standard Icelandic locale on his Linux system to produce a new locale with a collation order of Old West Norse words that feels natural for Swedish-speaking users.

Since all codepoints in the Unicode Basic Multilingual Plane have identical representations in UTF-16 and UTF-32, it is possible to come even closer to platform independence for applications that only work with such codepoints: In such cases, a facet that only accepts code points within the Basic Multilingual Plane exhibits the correct behaviour both for 16-bit and 32-bit `wchar_t`. For the particular application studied in this paper—the concordance program for Old West Norse and Runic Swedish—this approach works without flaws since all characters needed to represent Runic Swedish and Old West Norse are present within the Basic Multilingual Plane. There are, of course, settings where the approach outlined here leads to erroneous behaviour.

3.3 Towards true platform-independence

The fact that it was necessary to let the selection of code conversion facet depend on the platform in the solution proposed above is, if not inconvenient, at least esthetically unpleasing. One way to remedy this would be to define a data type, say `utf32_t`, guaranteed to hold every Unicode character and then define C++ strings using this data type as the underlying character type.

```
typedef std::basic_string<utf32_t> utf32_string;
```

At first, it seems straightforward to adapt the code conversion facet from Appendix A to this new setting: after all, the facet has the internal character type as a template parameter. However, a moment's reflection—or experimentation—indicates that this is not the case.

In order for the approach to work, it is necessary to define *character traits* for the new character type. Also, the fairly complex case conversion rules and

collation rules for Unicode strings must be implemented anew, since the standard C++ library only supports collation for the built-in character types. Consequently, facets for code conversion and collation that use the new character type `utf32_t` must also be implemented. Hence, it seems that truly platform-independent source code implementing code conversion, case conversion and collation comes only at a high price in terms of the amount of source code that must be written and maintained.

4 COLLATION AND CASE CONVERSION

Using the approach outlined in § 3.2, with the text represented internally as standard C++ wide strings, is is straightforward to perform collation: To collate the entries in the concordance, it is enough to supply the desired locale as a template parameter to the C++ multimap used to store the entries. Concretely, this data structure was initialized by the statement

```
std::multimap<std::wstring, Entry, std::locale>  concordance(loc);
```

where `loc` is the locale object created as described in § 3.2. Since only the code conversion facet of the locale had been replaced, the collation behaviour of the program is then defined by the surrounding operating system's locale. Similarly, the author used the following function object to conveniently convert C++ strings into lower case:

```
struct ToLower {
  ToLower(std::locale& l) : loc(l) {}
  wchar_t operator()(wchar_t c) { return std::tolower(c, loc); }
private:
  std::locale& loc;
};
```

An C++ wide string, say `word`, can then be transformed into lower case by the statement

```
std::transform(word.begin(), word.end(), word.begin(), ToLower(loc));
```

Also in this case, the program performs the conversion according to the system's locale. To be precise, the C++ standard states that the function `std::tolower()` "returns the corresponding lower-case character if it is known to exist, or its argument if not" [3, § 22.2.1.1.2, ¶ 10]. Therefore, it may happen that a character is not case converted. For instance, there are situations when case conversion of a single character produces multiple characters—the best known example is probably the German "sharp s": it is one character (ß) in lower-case but two characters (SS) in upper case. Clearly, the function `std::tolower()` is not adapted for cases like this and will most likely just return its argument for such problematic inputs.

5 CONCLUSIONS

This paper describes experiences drawn from the implementation of a concordance program for texts in arcane languages. The overall aim of the implementation was to write a program that used the standard C++ locale framework as much as possible. With only small amounts of platform-specific source code, it was possible to write a program that compiles cleanly both in modern Linux environments (Fedora Core 3 with gcc 3.4.2) and in Windows XP (with Visual Studio .NET 2003). The particular code written to supplement the standard C++ locale framework consisted only of a new code conversion facet that converts data between the external representation—fixed to UTF-8 on all platforms by design—and the internal representation—characters stored in the standard C++ type `wchar_t` according to the conventions of the operating system. In fact, it was possible to use the *same source code* in both Linux and Windows XP by making the assumption that the concordance program would never have to handle characters outside the Unicode Basic Multilingual Plane. Having introduced this assumption, a natural next step is to ask what other consequences and trade-offs the chosen implementation strategy implies.

On the positive side, the choice to rely on the standard C++ library functions to perform collation and case conversion freed us from the burden of implementing this functionality. Also, the user of the program can affect, e.g., the sorting order by selecting a proper locale. The fact that the code conversion facet used does not dependend on the locale selected by the user means that the the datafiles used by the concordance program will always be correctly interpreted, no matter the current default locale. This flexibility comes at a very low price—we only had to implement the proper code conversion facet. As mentioned above, it is easy to write a code conversion facet that correctly interprets characters in the Unicode Basic Multilingual Plane on platforms that use Unicode with either UTF-16 or UTF-32 for characters stored in a `wchar_t`. The C++ standard does not require that Unicode is used for characters of type `wchar_t`; it does in fact not even require `wchar_t` to hold 16 bits. Nevertheless, the assumption that most operating systems will indeed use Unicode encoded either with UTF-16 or UTF-32 for strings with `wchar_t` as the underlying character type seems to be a very weak one. The approach proposed in this paper therefore provides a way to write programs that are completely portable, with no change to the source code, given that they only process characters from the Unicode Basic Multilingual Plane.

On the negative side, some of the functionality required by the concordance program is available only if it is supported by the C++ runtime libraries on the particular platform where the program is compiled and executed. As an example, the author tried to compile the program with SunOneStudio 8 and then run the program on a Solaris 5.9 machine. This experiment did not turn out too well since the function used for case conversion could only case convert ASCII characters. Also, the collation did not work as expected in that environment.

Indeed, it cannot be guaranteed that the program produces identical output on all platforms since some aspects of the program's functionality is governed by the surrounding operating system. For a concordance program, it can be argued that it is actually good that the user can specify the collation order—an English-speaking user no doubt expects a different sorting order than an Icelandic-speaking user would. There are, of course, other applications where top priority is instead that the output is identical no matter the environment it was produced in. For such applications, dependence on library-provided functions and settings in the user's environment is not desirable.

The project described in this paper also says something about the usefulness of the C++ locale framework at large. It seems reasonable to expect that a program requiring fairly basic Unicode functionality can be implemented within the limits of standard C++. Reading and writing of characters should work well, and also character classification and simple string manipulation—such as manipulation, substitutions, concatenation—can be expected to work. However, already case conversion of arbitrary strings poses problems as the conversion of German sharp s from lower (ß) to upper (SS) case illustrates—case conversions where the number of characters changes falls outside of the standard C++ local model. For programs requiring more advanced Unicode features, such as conversion between different normalization forms and case conversion of arbitrary strings, the only possible route is probably to rely on some third-party library.

6 ACKNOWLEDGMENTS

The author wishes to thank Rune Palm for introducing him to Old West Norse, Runic Swedish, and the need of a concordance for the Scandinavian runic-text database.

References

1. Elmevik L, Peterson L. Samnordisk runtextdatabas. *Nytt om runer: Meldingsblad om runeforskning* 1993; 8:32 continued in 1997; 12:33–34.
2. ISO/IEC standard 8859-1:1998, *Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1*. ISO/IEC, 1998.
3. ISO/IEC standard 14882:2003, *Programming languages — C++* (2nd edn). ISO/IEC, 2003.
4. Langer A, Kreft K. *Standard C++ IOStreams and Locales.* Addison-Wesley, 2000.
5. Peterson L. Scandinavian runic-text data base: a presentation. In *The Twelfth Viking Congress: Developments Around the Baltic and the North Sea in the Viking Age*, Ambrosiani B, Clarke H (eds.), volume 3 of *Birka Studies*. Stockholm, 1994; 305–309.
6. Stroustrup B. *The C++ Programming Language* (special edn). Addison-Wesley, 2000.
7. The Unicode Standard, Version 4.0.0, defined by: *The Unicode Standard, Version 4.0*. Addison-Wesley, 2003.

A SOURCE CODE FOR THE CODE CONVERSION FACET

```
template<typename internT = wchar_t, typename externT = char>
class UTF8_UTF32 : public std::codecvt<internT, externT, mbstate_t>
{
  using std::codecvt<internT, externT, mbstate_t>::error;
  using std::codecvt<internT, externT, mbstate_t>::noconv;
  using std::codecvt<internT, externT, mbstate_t>::ok;
  using std::codecvt<internT, externT, mbstate_t>::partial;
protected:
  typename std::codecvt<internT, externT, mbstate_t>::result
  do_out(mbstate_t&,
         const internT* from, const internT* from_end,
         const internT*& from_next,
         externT* to, externT* to_end,
         externT*& to_next) const
  {
    from_next = from;
    to_next = to;

    while(from_next < from_end) {
      if(*from_next < 0x80) {
        if(to_next + 1 > to_end) break;
        *to_next++ = static_cast<externT>(*from_next++);
      }
      else if(*from_next < 0x00000800) {
        if(to_next + 2 > to_end) break;
        *to_next++ = static_cast<externT>(0xC0 | (*from_next >> 6));
        *to_next++ = static_cast<externT>(0x80 | (*from_next & 0x3F));
        from_next++;
      }
      else if(*from_next < 0x00010000) {
        if(*from_next >= 0xD800 && *from_next < 0xE000) return error;
        if(to_next + 3 > to_end) break;
        *to_next++ = static_cast<externT>(0xE0 | ( *from_next >> 12));
        *to_next++ = static_cast<externT>(0x80 | ((*from_next >>  6) & 0x3F));
        *to_next++ = static_cast<externT>(0x80 | ( *from_next        & 0x3F));
        from_next++;
      }
      else if(*from_next < 0x00110000) {
        if(to_next + 4 > to_end) break;
        *to_next++ = static_cast<externT>(0xF0 | ( *from_next >> 18));
        *to_next++ = static_cast<externT>(0x80 | ((*from_next >> 12) & 0x3F));
        *to_next++ = static_cast<externT>(0x80 | ((*from_next >>  6) & 0x3F));
        *to_next++ = static_cast<externT>(0x80 | ( *from_next        & 0x3F));
        from_next++;
      }
      else {
        return error;
      }
    }
    return (from_next == from_end) ? ok : partial;
  }
```

```
typename std::codecvt<internT, externT, mbstate_t>::result
do_unshift(mbstate_t&,
           externT* to, externT*,
           externT*& to_next) const
{
  to_next = to;
  return noconv;
}

typename std::codecvt<internT, externT, mbstate_t>::result
do_in(mbstate_t&,
      const externT* from, const externT* from_end,
      const externT*& from_next,
      internT* to, internT* to_end,
      internT*& to_next) const
{
  from_next = from;
  to_next = to;

  while(to_next < to_end && from_next < from_end) {
    if((*from_next & 0x80) == 0x00) {
      *to_next++ = *from_next++;
    }
    else if((*from_next & 0xE0) == 0xC0) {
      if(from_next + 2 > from_end) break;
      *to_next  = (*from_next++ & 0x1F) << 6;
      if((*from_next & 0xC0) != 0x80) return error;
      *to_next |=  *from_next++ & 0x3F;
      if(*to_next < 0x80) return error;
      to_next++;
    }
    else if((*from_next & 0xF0) == 0xE0) {
      if(from_next + 3 > from_end) break;
      *to_next  = (*from_next++ & 0x0F) << 12;
      if((*from_next & 0xC0) != 0x80) return error;
      *to_next |= (*from_next++ & 0x3F) <<  6;
      if((*from_next & 0xC0) != 0x80) return error;
      *to_next |=  *from_next++ & 0x3F;
      if(*to_next < 0x00000800) return error;
      if(*to_next >= 0xD800 && *to_next < 0xE000) return error;
      to_next++;
    }
    else if((*from_next & 0xF8) == 0xF0) {
      if(from_next + 4 > from_end) break;
      *to_next  = (*from_next++ & 0x07) << 18;
      if((*from_next & 0xC0) != 0x80) return error;
      *to_next |= (*from_next++ & 0x3F) << 12;
      if((*from_next & 0xC0) != 0x80) return error;
      *to_next |= (*from_next++ & 0x3F) <<  6;
      if((*from_next & 0xC0) != 0x80) return error;
      *to_next |=  *from_next++ & 0x3F;
      if(*to_next < 0x00010000 || *to_next >= 0x00110000) return error;
      to_next++;
    }
```

```
      else {
        return error;
      }
    }
    return (from_next == from_end) ? ok : partial;
  }

  int do_encoding() const throw() { return 0; }

  bool do_always_noconv() const throw() { return false; }

  int do_length(mbstate_t&, const externT* from,
                const externT* end, size_t max) const
  {
    int len = 0;
    while(from < end && static_cast<size_t>(len) < max) {
      if((*from & 0x80) == 0x00) {
        from++;
      }
      else if((*from & 0xE0) == 0xC0) {
        if(from + 2 > end || (from[1] & 0xC0) != 0x80) break;
        from += 2;
      }
      else if((*from & 0xF0) == 0xE0) {
        if(from + 3 > end ||
           (from[1] & 0xC0) != 0x80 ||
           (from[2] & 0xC0) != 0x80) break;
        from += 3;
      }
      else if((*from & 0xF8) == 0xF0) {
        if(from + 4 > end ||
           (from[1] & 0xC0) != 0x80 ||
           (from[2] & 0xC0) != 0x80 ||
           (from[3] & 0xC0) != 0x80) break;
        from += 4;
      }
      else {
        break;
      }
      len++;
    }
    return len;
  }

  int do_max_length() const throw() { return 4; }
};
```