C SC22/WG14 N0379 X3J11/94-0064
C++ SC22/WG21 N0543 X3J16/94-0156

October 20, 1994

Internationalization:

A Primer for C/C++ Programmers

Rex Jaeschke, Chair X3J11
2051 Swans Neck Way
Reston, VA 22091
USA
Telephone: +1 703 860-0091
E-mail: rex@aussie.com

Contents

1 Introduction

Prior to the standardization of C, C programs executed in what amounted to as a "USA-English" environment. For example, `isupper` returned true only for the Roman letters A–Z, the decimal-point character printed by `printf` was the period, and dates were formatted as mm/dd/yy. As a result of standardization, the behavior of various standard library functions is now described in terms of the execution environment, which can be changed at runtime. For example, the `is*` and `to*` functions in `<ctype.h>` can properly handle local characters, the `printf` and `scanf` families can deal with decimal points other than a period, and local date and time formats are permitted.

A major addition to C was the ability to allow characters in other alphabets to be used inside character constants, comments, header names, and string literals. (However, letters in identifiers are still restricted to the Roman alphabet and arithmetic constants must use digits from the Arabic number system.) A handful of new functions were defined to deal with interpreting and/or converting multibyte and wide characters.

Soon after the initial ISO C Standard was published, work began on an amendment to that standard. The primary purpose of this amendment was to provide further support for internationalization. This support was achieved via new headers, which contain macros, type definitions, and functions. The amendment is known officially as ISO C Amendment 1.

This article describes only the support Standard C provides for achieving internationalization. It does not attempt to describe a strategy for achieving internationalization.

2 Definition of Terms

Before we can discuss internationalization in detail or even look at programming examples, it is necessary to define a number of terms, many of which are interrelated.

Byte: The unit of data storage large enough to hold any member of the basic execution character set.

Category: One of a number of components that, when combined, describe a locale.

char: An integral type whose range of values can represent distinct codes for all members of the basic execution character set. The `sizeof` operator produces the size of its operand, measured in bytes. Since `sizeof(char)` is 1 by definition, an object of type `char` occupies exactly one byte.

Character: A bit representation that fits in a byte. Each member of the basic source and basic execution character sets must fit into a byte. [Unfortunately, the term 'character' has been used to mean different things to different audiences. In the context of the C Standard, it is used to mean 'single-byte character.' It is likely that C's definition of this term will be clarified in the next revision of the C Standard.]

Character constant: A sequence of one or more multibyte characters enclosed in single-quotes, as in `'x'`.

Comment: A sequence of zero or more multibyte characters enclosed between `/*` and `*/`.

Encoding scheme: A set of rules for parsing a stream of bytes into a group of coded characters. The meaning of a multibyte character can vary if the encoding scheme provides state-dependent encoding via shift sequences. Standard C requires that no encoding scheme have a byte with value zero as the second or subsequent byte of a multibyte character. This restriction allows many of the traditional string manipulation functions to be used transparently with strings containing multibyte characters.

Encoding schemes for wide characters are quite simple—all characters have the same internal width and each character has a unique value. (Note, however, that in Unicode, some characters, such as accented letters not given their own alternate encoding, are encoded as two wide characters: the letter, followed by the accent.)

Examples of encoding schemes are: ASCII, EBCDIC, EUC, JIS, Shift-JIS, Unicode, 10646.UCS-2, and 10646.UCS-4.

Execution character set: The set of characters available for use during execution of a program. The basic execution character set must include all of the characters in the basic source character set, the null character, and control characters representing alert, backspace, carriage return, and new line. The members of the extended execution character set are locale-specific and may be single-byte or multibyte characters.

Header name: A sequence of one or more multibyte characters that identifies the name of a header used as the subject of a `#include` preprocessor directive.

Locale: A set of conventions based on some nationality, culture, or language. A locale is made up of a set of categories.

Locale-specific behavior: Behavior pertaining to a particular locale. Examples include the decimal-point character and date and time formats.

Mixed locale: A composite locale comprised of a set of conventions representing two or more distinct nationalities, cultures, or languages. For example, a Swiss locale might include some German aspects as well as some French and/or Italian aspects.

Multibyte character: A character from the execution character set that *may* require more than a single byte for its representation. Examples include character sets that accommodate Japanese, Chinese, Korean, and Arabic alphabets. Multibyte characters can appear in character constants, comments, header names, and string literals.

Single-byte characters (from the ASCII and EBCDIC characters sets, for example) are multibyte characters as well; they just happen to fit in one byte!

Multibyte character string: A sequence of zero or more multibyte characters terminated by, and including, a null character.

Null character: A byte with all bits set to zero.

Null wide character: A wide character with all bits set to zero.

Shift sequence: A sequence of one or more bytes that indicate a change in encoding. When a sequence of multibyte characters is being scanned, the detection of a shift sequence causes the characters following to be interpreted differently until the shift state is changed (possibly by restoring to the original state) or the end of the character sequence is reached. All comments, string literals, character constants, and header names are required to begin and end in their initial shift state. In the initial shift state, single-byte characters have their usual meaning; that is, they do not alter the shift state. A redundant shift sequence is one that is followed immediately by another shift sequence or is one that switches to the (already) current mode.

Single-byte character: A character from the execution character set that can be represented in a single byte. Examples of character sets made up entirely of single-byte characters are ASCII and EBCDIC.

Source character set: The set of characters available for use in writing source code. The basic source character set must include the 52 upper- and lowercase letters from the English alphabet, the digits 0–9, the graphic characters `! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ _ { | } ~`, the space character, and control characters representing horizontal tab, vertical tab, and form feed. If any other characters are seen except inside character constants, comments, header names, and string literals, the behavior is undefined. The extended source character set may include other single- and/or multibyte characters.

String: A sequence of zero or more multibyte characters terminated by, and including, a null character.

String literal: A sequence of zero or more multibyte characters enclosed in double-quotes, as in `"xyz"`.

wchar_t: An integral type whose range of values can represent distinct codes for all members of the largest extended execution character set specified among the supported locales. This is the type of a wide character. If all character sets used are single-byte, an implementation may define `wchar_t` as a synonym for `char`.

Wide character: An object capable of representing distinct codes for all members of the largest extended execution character set specified among the supported locales. A wide character has type `wchar_t`. Unlike a multibyte character, each wide character takes up the same amount of storage. One character set based on wide characters is Unicode, a subset of the code defined by ISO 10646. When represented as a wide character, the null character shall have the code value zero.

Wide character constant: A sequence of one or more multibyte characters enclosed in single-quotes and prefixed with `L`, as in `L'x'`. Despite its name, a wide character constant is written using multibyte characters rather than wide characters because source programs are composed of multibyte characters rather than wide characters. During translation, each multibyte character in the constant will be mapped to a corresponding wide character.

Wide character string: A sequence of zero or more wide characters terminated by, and including, a null wide character.

Wide character string literal: A sequence of zero or more multibyte characters enclosed in double-quotes and prefixed with `L`, as in `L"xyz"`. Despite its name, a wide character string literal is written using multibyte characters rather than wide characters because source programs are composed of multibyte characters rather than wide characters. During translation, each multibyte character in the string literal will be mapped to a corresponding wide character.

3 Locales

Two locales are defined by Standard C: `"C"` and the implementation-defined native locale `""`. (An implementation that provides the bare minimum locale support will provide locale `"C"` and it will make the native locale `""` be the same as locale `"C"`.) Any number of other locales are permitted, with arbitrary names.

At program startup, the default locale is `"C"`. Changing one or more categories of a locale to something other than `"C"` requires a call to the function `setlocale`, declared in `<locale.h>`.

3.1 A Simple Example

Consider the following program that performs a number of locale-specific operations based on a user-specified locale:

```
#include <locale.h>
#include <stdio.h>
#include <ctype.h>
#include <time.h>

main()
{
        char locale[31];
        char *ploc_str;
        int c;
        time_t system_time;
        char time_text[81];

        printf("Enter locale name: ");
/*1*/   scanf("%30s%*c", locale);
```

```
/*2*/   ploc_str = setlocale(LC_ALL, locale);
        if (ploc_str == NULL) {
                printf("Can't establish locale \"%s\"\n", locale);
                return 0;
        }

        printf("Established locale \"%s\"\n", locale);

        printf("Enter a single character: ");
/*3*/   c = getchar();

/*4*/   printf("Character '%c' %s alphabetic\n", c,
                isalpha(c) ? "is" : "is not");

/*5*/   printf("The value '12 point 345' is written as %.3f\n", 12.345);

        system_time = time(NULL);

/*6*/   strftime(time_text, sizeof(time_text), "%x %A %B %d\n",
                localtime(&system_time));
        printf(time_text);

        return 0;
}
```

Here are several sets of input and their corresponding output:

```
Enter locale name: american
Established locale "american"
Enter a single character: a
Character 'a' is alphabetic
The value '12 point 345' is written as 12.345
07/03/94 Sunday July 03

Enter locale name: french
Established locale "french"
Enter a single character: ê
Character 'ê' is alphabetic
The value '12 point 345' is written as 12,345
03/07/94 dimanche juillet 03

Enter locale name: german
Established locale "german"
Enter a single character: ä
Character 'ä' is alphabetic
The value '12 point 345' is written as 12,345
03.07.94 Sonntag Juli 03

Enter locale name: spanish
Established locale "spanish"
Enter a single character: ñ
Character 'ñ' is alphabetic
The value '12 point 345' is written as 12,345
03/07/94 domingo julio 03
```

```
Enter locale name: swedish
Established locale "swedish"
Enter a single character: å
Character 'å' is alphabetic
The value '12 point 345' is written as 12,345
1994-07-03 söndag juli 03
```

In case 1, the locale name is entered and the terminating new-line discarded. In the implementation used to build the examples in this article (Microsoft's Windows NT), 'american', 'french', 'german', 'spanish', and 'swedish' are valid locale names.

Since the category macro `LC_ALL` is used, the call to `setlocale` in case 2, attempts to establish the user-specified locale for all locale categories. If this is not possible, a null pointer is returned.

In case 3, `getchar` is used to read a single character from input. Since `isalpha` is locale-specific, letters other than Roman A–Z and a–z are accepted, as shown by the output from case 4.

The output from case 5 shows that `printf` is also locale-specific; in some countries, the comma is used as the decimal-point character.

The function `strftime` provides date and time information formatted in a locale-specific manner. As shown, the day, month, and year order and separators can vary from one country to another. Also, users can get day and month names in their native language.

3.2 Approaches to Using Locales

There are three ways in which to use locales:

1. Never call `setlocale` at all. By default, all processing will be done using locale `"C"`, giving us a "USA-English" mode of operation. This approach is simple; it's what we've had these past 20+ years.

2. At the start of `main`, call `setlocale` once to establish all categories to the native locale `""`, whatever that may be. This approach is fairly straight-forward; all locale-specific functions do their thing and we work in (presumably) our native environment. However, we should consider the case where a program built using this approach is run in an environment where the native locale is different. Will the program behave in a reasonable manner or did we hard-code something that really is locale-specific?

3. Use multiple locales by switching some or all categories back and forth from one locale to another. This approach requires more care since we must keep careful track of the current mode of each locale category as we go, making sure to not make unreasonable assumptions along the way. Another concern is whether all locales are known in advance to the programmer or whether one or more are supplied at runtime.

3.3 Locale Names

The spelling of a locale name is unspecified by Standard C; a locale name is simply an implementation-defined string. As such, it may contain multibyte characters. Consider the case of a locale designed to support data processing in Germany. A few possible spellings for that locale's name are `"german"`, `"deutsch"`, `"deitser"`, `"alemán"`, and `"allemand"`, along with their uppercase equivalents or with leading capital letters. It depends on programmers or users cultural background and native language as to which form they would prefer or expect to see or use.

Hard-coding locale names is probably a bad idea, particularly if portability is an issue. If portability is not an issue, you might question this advice. However, consider the case where your implementation does not support a locale that you need. If that implementation provides a way to add locales the locale may come from another vendor or user. And its name may be out of your control. Of course, if that implementation does not provide a way to integrate third-party locales, you may have to move to one that does, and that is a port!

In any event, for the purposes of this discussion we will use a header called `"locnames.h"` which contains macros whose names are of the form `LOC_*`. For example:

```
#define LOC_American    "american"
#define LOC_Arabic      "arabic"
        ...
#define LOC_French      "french"
#define LOC_German      "german"
        ...
#define LOC_Spanish     "spanish"
#define LOC_Swiss       "swiss"
```

This user-defined and maintained header can contain conditional-compilation directives as necessary, to accommodate different locale name spellings.

Clearly, it is useful to hide the real locale name spelling from the programmer. However, when the value of a locale name macro is displayed directly, the user will see the underlying spelling. If that is undesirable, the name of the macro can be displayed instead, as follows:

```
#include <locale.h>
#include "locnames.h"
#include <stdio.h>

main()
{
        char *ploc_str;

        ploc_str = setlocale(LC_ALL, LOC_German);
        if (ploc_str == NULL) {
                printf("Can't establish locale %s\n", STR(LOC_German));
                return 0;
        }

        printf("Established locale %s\n", STR(LOC_German));

        return 0;
}
```

The macro STR is defined in "locnames.h" using:

```
#define STR(x) #x
```

The possible outputs now become:

```
Established locale LOC_German

Can't establish locale LOC_German
```

On some systems it may be possible to get locale names from an external source, such as via environment variables.

At the time of writing, at least one group was known to be working on a registry of standard locale names.

3.4 Locale Categories

A locale can be established for all locale-specific operations or for just a subset, depending on the category specified in a call to setlocale. Standard C defines six category macros in <locale.h>:

LC_ALL: This category includes all other categories.

LC_COLLATE: This category affects the behavior of the strcoll and strxfrm functions.

`LC_CTYPE`: This category affects the behavior of the character handling functions with the exception of `isdigit` (which are not locale-dependent) and `isxdigit`. It also affects the behavior of the multibyte functions.

`LC_MONETARY`: This category affects the monetary formatting information returned by the function `localeconv`.

`LC_NUMERIC`: This category affects the decimal-point character for the formatted I/O functions (such as `printf` and `scanf`) and the string conversion functions (such as `atof` and `strtod`), as well as the non-monetary formatting information returned by the function `localeconv`.

`LC_TIME`: This category affects the behavior of the `strftime` function.

These macros expand to integral constant expressions having distinct values. An implementation may define additional category macros provided their names begin with the characters `LC_` followed by an uppercase letter. (The POSIX standard defines a category called `LC_MESSAGES` which allows the user to find out the spelling of the affirmative and negative response to a yes/no question.)

A mixed locale can be established by calling `setlocale` a number of times, once per category. For example:

```
setlocale(LC_COLLATE, LOC_German);
setlocale(LC_CTYPE, LOC_French);
setlocale(LC_NUMERIC, LOC_Italian);
```

3.5 The `setlocale` Function

When a call to `setlocale` succeeds, it returns a pointer to the string associated with the specified locale name and category combination. Let us call such a string a *locale string*. The format of a locale string is unspecified by Standard C. The following program displays several such locale strings:

```
#include <stdio.h>
#include <locale.h>
#include "locnames.h"

main()
{
/*1*/   printf("|%s|\n", setlocale(LC_ALL,   NULL));
/*2*/   printf("|%s|\n", setlocale(LC_CTYPE, LOC_French));
/*3*/   printf("|%s|\n", setlocale(LC_TIME,  NULL));
/*4*/   printf("|%s|\n", setlocale(LC_ALL,   NULL));

        return 0;
}
```

The output produced on one implementation that supported a French locale was:

```
|C|
|French_France.850|
|C|
|LC_COLLATE=C;LC_CTYPE=French_France.850;LC_MONETARY=C;LC_NUMERIC=C;LC_TIME=C|
```

When `setlocale` is called with a locale name of `NULL`, the locale string for the specified category in the current locale is returned. The category is not changed. Since the default locale at program startup is "C", the output produced in cases 1 and 3 is not surprising. However, since the format of a locale string is unspecified there is no guarantee that any other implementation will produce this result.

The output produced in cases 2 and 4 is, of course, specific to the implementation. When case 4 is executed, a mixed locale has already been established. Therefore, the locale string returned in this case must somehow include a description of that mixed locale.

The locale string produced for a given locale name and category combination can be used as the locale name in a subsequent call to `setlocale` for the same category. For example, in the following program, the function `handle_dates` temporarily switches to Spanish date/time processing and then restores the original mode:

```
void handle_dates()
{
        char *ploc_str;
        char *ploc_save;

        ploc_str = setlocale(LC_TIME, NULL);
/*1*/   ploc_save = malloc(strlen(ploc_str) + 1);
/*2*/   strcpy(ploc_save, ploc_str);
        printf("Saved current LC_TIME\n");

        setlocale(LC_TIME, LOC_Spanish);
        printf("Established %s LC_TIME\n", STR(LOC_Spanish));

        printf("Doing %s-specific date/time processing\n", STR(LOC_Spanish));

        setlocale(LC_TIME, ploc_save);
        printf("Restored saved LC_TIME\n");
        free(ploc_save);
}
```

Assuming `setlocale` and `malloc` don't fail, the output produced is:

```
Saved current LC_TIME
Established LOC_Spanish LC_TIME
Doing LOC_Spanish-specific date/time processing
Restored saved LC_TIME
```

The memory allocated for a locale string is managed by the library. Since it may be recycled in subsequent calls to `setlocale`, it is the user's responsibility to make a copy for later use, as shown in cases 1 and 2.

There is no way to identify which locale is currently established. Since the format of a locale string is unspecified there is no guarantee that the locale strings produced by two identical calls to `setlocale`, will compare equal.

3.6 `struct lconv` and the `localeconv` Function

The header `<locale.h>` contains a definition for the type `struct lconv` whose members provide access to information regarding the formatting of monetary and non-monetary numeric values.

The members having type `char *` are pointers to strings. Any of the pointer members except `decimal_point` can point to an empty string, which indicates the value is not available in the current locale or is of zero length. In the `"C"` locale, the decimal-point character is a period and all other pointer members point to an empty string.

The members having type `char` contain nonnegative values. If a value of `CHAR_MAX` is present, it indicates that the real value is not available in the current locale. In the `"C"` locale, these members all have value `CHAR_MAX`.

The structure may contain other members. The ordering of members is unspecified. The following members must be present:

`char *decimal_point`: The decimal-point character used to format non-monetary quantities.

`char *thousands_sep`: The character used to separate groups of digits before the decimal-point character in formatted non-monetary quantities.

`char *grouping`: A string whose elements indicate the size of each group of digits in formatted non-monetary quantities.

`char *int_curr_symbol`: The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol. The fourth character is the character used to separate the international currency symbol from the monetary quantity. The fifth character is a null character.

`char *currency_symbol`: The local currency symbol applicable to the current locale.

`char *mon_decimal_point`: The decimal-point used to format monetary quantities.

`char *mon_thousands_sep`: The separator for groups of digits before the decimal-point character in formatted monetary quantities.

`char *mon_grouping`: A string whose elements indicate the size of each group of digits in formatted monetary quantities.

`char *positive_sign`: The string used to indicate a nonnegative-valued formatted monetary quantity.

`char *negative_sign`: The string used to indicate a negative-valued formatted monetary quantity.

`char int_frac_digits`: The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

`char frac_digits`: The number of fractional digits (those after the decimal-point) to be displayed in a locally formatted monetary quantity.

`char p_cs_precedes`: Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.

`char p_sep_by_space`: Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity.

`char n_cs_precedes`: Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative formatted monetary quantity.

`char n_sep_by_space`: Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

`char p_sign_posn`: Set to a value indicating the positioning of the `positive_sign` for a nonnegative formatted monetary quantity.

`char n_sign_posn`: Set to a value indicating the positioning of the `negative_sign` for a negative formatted monetary quantity.

The elements of `grouping` and `mon_grouping` are interpreted according to their value, as follows:

`CHAR_MAX`: No further grouping is to be performed.

0: The previous element is to be repeatedly used for the remainder of the digits.

other: The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

For example, a string of `"\3"` consists of the value 3 followed by a null character (having value 0). This results in digits being grouped in lots of 3, as in 999 999 999. A string containing `"\2\3\4"` results in the digit grouping 9999 999 99.
The value of `p_sign_posn` and `n_sign_posn` is interpreted according to the following:

0: Parentheses surround the quantity and `currency_symbol`.

1: The sign string precedes the quantity and `currency_symbol`.

2: The sign string succeeds the quantity and `currency_symbol`.

3: The sign string immediately precedes the `currency_symbol`.

4: The sign string immediately succeeds the `currency_symbol`.

When `localeconv` is called, a structure of type `struct lconv` is created and initialized with values corresponding to the current locale. The address of that structure is returned to the caller. This structure may be overwritten by future calls to `localeconv` or by calls to `setlocale` that refer to the categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC`. (Note, however, that saving a copy of this structure is insufficient to save the complete numeric formatting description since the strings pointed to by the `char *` members might also be overwritten by subsequent calls to those functions.)

The following example shows a call to `localeconv` and the subsequent display of two of the structure members' contents.

```
#include <stdio.h>
#include <locale.h>
#include "locnames.h"

main()
{
        struct lconv *pst;

        setlocale(LC_ALL, LOC_French);
        pst = localeconv();

        printf("Locale: %s\n", STR(LOC_French));
        printf("decimal_point:   %s\n", pst->decimal_point);
        printf("int_curr_symbol: %s\n", pst->int_curr_symbol);

        return 0;
}
```

The output produced is:

```
Locale: LOC_French
decimal_point:   ,
int_curr_symbol: FRF
```

3.7 The `strcoll` Function

This function works just like `strcmp` except that `strcoll` uses a collating sequence established via the `LC_COLLATE` category. For example:

```
#include <stdio.h>
#include <string.h>
#include <locale.h>
#include "locnames.h"

main()
{
        char str1[6], str2[6];
        int i;
```

```
        if (setlocale(LC_ALL, LOC_French) == NULL) {
                printf("Can't establish French locale\n");
        }

        printf("Enter two strings: ");
        scanf("%5s %5s", str1, str2);

        i = strcoll(str1, str2);
        if (i == 0) {
                printf("The strings are the same\n");
        }
        else if (i < 0) {
                printf("%s < %s\n", str1, str2);
        }
        else {
                printf("%s > %s\n", str1, str2);
        }

        return 0;
}
```

Several inputs and their corresponding outputs are:

```
Enter two strings: e ê
e < ê

Enter two strings: ê f
ê < f
```

As shown by the outputs, ê sorts between e and f, just as the French would want.

4 Multibyte Characters

One of the earliest, and probably the biggest, markets for multibyte character support is in Japan. Therefore, examples in this section will be based on encoding schemes for Japanese text processing. An understanding of these schemes makes it easier to understand other schemes.

In Japan, a single text message can be composed of characters from four different writing systems: Kanji, Hiragana, Katakana, and Roman. Kanji number in the tens of thousands and are represented by pictures. Hiragana and Katakana (collectively known as kana) are syllabaries each containing about 80 sounds which are also represented by pictures. The Roman characters include some 95 letters, digits, and punctuation marks. One-byte control characters may also be present. I/O of non-Roman characters is a complicated issue and is also outside the scope of this discussion; we're only interested in internal representation.

The number of bytes required to represent different members of the same execution character set can vary. For example, some members might require one byte, some two, and others, three or more. The representation of a multibyte character is determined by its encoding scheme and more than one encoding scheme may be available for a given multibyte character set. For any supported locale, the maximum number of bytes needed to represent a multibyte character, including shift state information, is available via the macro `MB_LEN_MAX`. This macro is defined in `<limits.h>`. For the current locale, the maximum number of bytes needed to represent a multibyte character, including shift state information, is available via the macro `MB_CUR_MAX`. This macro is defined in `<stdlib.h>`.

Why have different widths for characters in the same character set? Why not simply make all characters the same width? These are important questions. A simple answer is "To save space." If a text file contains many characters that can be represented each in a single byte, that file will be smaller than if all characters were stored as two- or four-byte characters. A common example of this would be program source files which, for the most part, contain Roman characters. Once in memory, however, it is easier to manipulate characters

if they all have the same width. Such characters are known as wide characters and will be discussed in the next section.

There is no universally recognized multibyte encoding scheme for Japanese. However, three schemes are prominent. They are JIS, Shift-JIS, and EUC. (JIS is an abbreviation for "Japanese Industrial Standard" while EUC stands for "Extended UNIX Code.") JIS is the simplest scheme. It uses 7 bits of each byte and requires escape sequences to shift between one- and two-byte modes. Shift-JIS and EUC use 8 bits of each byte and the value of the first byte is used to determine whether a second byte follows.

4.1 JIS Encoding

JIS supports a number of standard Japanese character sets, some requiring one byte, others two. The following table shows some of these character sets and the shift sequences needed to establish them within a JIS-encoded file or string:

JIS Shift Sequences	
Character Set	*Shift Sequence*
JIS C 6226-1978	<ESC> $ @
JIS X 0208-1983	<ESC> $ B
JIS X 0208-1990	<ESC> & @ <ESC> $ B
JIS X 0212-1990	<ESC> (D
JIS-Roman	<ESC> (J
ASCII	<ESC> (B

The first four character sets are two-byte while the last two sets are one-byte. <ESC> represents the 'escape' character having value 0x1B.

Consider the following character sequence:

The Kanji <ESC>$B...some kanji...<ESC>(Bspell 'Tokyo.'

At the start of the sequence we are in some initial shift state. Let's assume that is ASCII. Therefore, characters are assumed to be one-byte ASCII codes until the shift sequence <ESC>$B is seen. This switches us to two-byte mode as defined by JIS X 0208-1983. The shift sequence <ESC>(B switches us to ASCII mode, which also happens to be the initial shift state. The sequence used in this example might well be a string literal used in a source file, in a call to `printf` or `strcpy`, for example.

Encoding schemes that use shift sequences are not very efficient for internal storage or processing. For example, the shift sequence for JIS X 0208-1990 requires six bytes. Frequent switching between character sets in a file or string could result in the number of bytes used in shift sequences exceeding that used to represent actual data!

Consider the following character sequences:

Hello <ESC>$B<ESC>(Bthere.
Hello <ESC>(Bthere.

In the first case, we see one shift sequence immediately followed by another, resulting in the first shift sequence being redundant. In the second case, we see the mode set to the one that is already current, again making the shift sequence redundant. Of course, there can be an arbitrary number of redundant shift sequences in a file or string, all of which take up space but contribute nothing useful.

Encodings containing shift sequences are used primarily as an *external* code: one that allows information interchange between a program and the outside world. For example, many communications paths are limited to 7 bits. Therefore, sending electronic mail containing multibyte Japanese characters over such paths requires an encoding scheme that uses only 7 bits of each byte.

4.2 Shift-JIS Encoding

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

1. Any byte having a value in the range 0x21–7E is assumed to be a one-byte ASCII/JIS-Roman character.
2. Any byte having a value in the range 0xA1–DF is assumed to be a one-byte half-width katakana character. (This character set will not be discussed further.)
3. Any byte having a value in the range 0x81–9F or 0xE0–EF is assumed to be the first byte of a two-byte character from the set JIS X 0208-1990. The second byte must have a value in the range 0x40–7E or 0x80–FC.

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

Shift-JIS was invented by Microsoft. And since Microsoft has been a significant player in the field of internationalization, this encoding scheme has become very popular.

4.3 EUC Encoding

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

1. Any byte having a value in the range 0x21–7E is assumed to be a one-byte ASCII/JIS-Roman character.
2. Any byte having a value in the range 0xA1–FE is assumed to be the first byte of a two-byte character from the set JIS X 0208-1990. The second byte must also have a value in that range.
3. Any byte having the value 0x8E is assumed to be followed by a second byte with a value in the range 0xA1–DF, which represents a half-width katakana character.
4. Any byte having the value 0x8F is assumed to be followed by a two more bytes with values in the range 0xA1–FE, which together represent a character from the set JIS X 0212-1990.

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

4.4 Shift States

For encodings containing shift sequences, the functions `mblen`, `mbtowc`, and `wctomb` are required to maintain information about the current shift state. However, if the `LC_CTYPE` category is changed for a given locale, the current shift state becomes indeterminate. The multibyte functions can be put into their initial shift state by a call with a value of `NULL` for the `char *` argument. For example:

```
#include <stdlib.h>
#include <stdio.h>

main()
{
        int i;

/*1*/   i = mblen(NULL, 0);
/*2*/   printf("There %s a state-dependent encoding\n",
                (i != 0) ? "is" : "is not");

        return 0;
}
```

In case 1, we set `mblen` to its initial shift state by passing it a null pointer. The value returned is nonzero if the encoding scheme currently in use uses shift sequences, zero otherwise. The four other multibyte functions behave likewise when given a null pointer argument.

`mbstowcs` assumes that the multibyte string it is to convert starts in the initial shift state. When `wcstombs` creates a multibyte string it first sets to the initial shift state.

5 Wide Characters

As stated earlier, multibyte encoding provides an efficient way for moving characters around outside programs and between programs and the outside world. Once inside a program, it is easier and more efficient to deal with characters that have the same size and format. Wide characters serve this purpose.

Consider a filename string containing a directory path where adjacent names are separated by a slash. (/root/counts/test.dat, for example.) To find the actual filename, in a single-byte character string we can start at the back of the string. When we find the first separator we know where the filename proper starts. However, if the string contains multibyte characters, we must scan from the front so we don't inspect bytes out of context. If, however, the string contains wide characters, we can scan from the back.

A number of wide character standards exist or are emerging. The most well-known ones that support characters larger than eight bits are ISO 10646.UCS-2, ISO 10646.UCS-4, and Unicode. The UCS-2 encoding is based on 16-bit characters and exactly matches Unicode. The UCS-4 encoding uses 32-bit characters and is still under development.

Conceptually, we can think of wide character sets as being extended ASCII or EBCDIC; each unique character is assigned a distinct value.

Since the size of a wide character is not universally fixed, the type synonym `wchar_t` was invented as the wide character type. This synonym is defined in `<stddef.h>` and `<stdlib.h>`.

Standard C allows wide characters in character constants and strings literals provided new notation is used. For example:

- `L'a'` is a wide character constant having type `wchar_t`.
- `L"abc"` is a wide character string having type `wchar_t [4]`.

Each member of the basic execution and source character sets shall have a code value equal to its value when used as the lone character in an integer character constant. For example, this requires that `'a' == L'a'` be true.

6 Multibyte Functions

Standard C includes five new functions to deal with interpreting and/or converting multibyte and wide characters. These functions are referred to as the *multibyte functions*. They are: `mblen`, `mbstowcs`, `mbtowc`, `wcstombs`, and `wctomb`, all declared in `<stdlib.h>`.

6.1 The `mblen` Function

`mblen` reports the number of bytes in a given multibyte character. For example:

```
#include <stdlib.h>
#include <stdio.h>

main()
{
/*1*/   printf("MB_CUR_MAX = %lu\n", (unsigned long)MB_CUR_MAX);
/*2*/   printf("i = %d\n", mblen("", MB_CUR_MAX));
/*3*/   printf("i = %d\n", mblen("ABCDEFGHIJ", MB_CUR_MAX));

        return 0;
}
```

The output produced on one system was:

```
MB_CUR_MAX = 1
i = 0
i = 1
```

The output from case 1 indicates that the program executed in single-byte mode. In case 2, the multibyte string is empty so `mblen` returns zero. In case 3, the multibyte string contains a number of multibyte characters; however, only the first is processed resulting in `mblen` returning one. If the first `MB_CUR_MAX` bytes do not form a valid multibyte character, `mblen` returns -1.

Since `mblen` might be called repeatedly on successive multibyte characters in a string, it must remember the current shift state.

6.2 The `mbtowc` Function

`mbtowc` converts a given multibyte character to its corresponding wide character. For example:

```
#include <stdlib.h>
#include <stdio.h>

main()
{
        int i;
        wchar_t wc;

        printf("MB_CUR_MAX = %lu\n", (unsigned long)MB_CUR_MAX);

/*1*/   i = mbtowc(&wc, "", MB_CUR_MAX);
        printf("i = %d, wc = %#41X\n", i, (unsigned long)wc);

/*2*/   i = mbtowc(&wc, "ABCD", MB_CUR_MAX);
        printf("i = %d, wc = %#41X\n", i, (unsigned long)wc);

        return 0;
}
```

The output produced on one ASCII-based system was:

```
MB_CUR_MAX = 1
i = 0, wc =    0
i = 1, wc = 0X41
```

The value returned by `mbtowc` has the same meaning as for `mblen`; if the multibyte string is empty, zero is returned and the null character is converted to a wide null character which also has value zero. If the multibyte string contains a number of multibyte characters, only the first is processed, in this case, the letter A (ASCII value 41 hex). If the first `MB_CUR_MAX` bytes do not form a valid multibyte character, -1 is returned.

Since `mbtowc` might be called repeatedly on successive multibyte characters in a string, it must remember the current shift state.

6.3 The `wctomb` Function

`wctomb` converts a given wide character to its corresponding multibyte character. For example:

```
#include <stdlib.h>
#include <stdio.h>

main()
{
        int i, j;
        char string[10];

        printf("MB_CUR_MAX = %lu\n", (unsigned long)MB_CUR_MAX);

/*1*/   i = wctomb(string, L'A');
        printf("i = %d, string = ", i);

        for (j = 0; j < i; ++j) {
                printf(" %#4X", string[j]);
        }
        putchar('\n');

        return 0;
}
```

The output produced on one ASCII-based system was:

```
MB_CUR_MAX = 1
i = 1, string = 0X41
```

If the wide character value has no corresponding multibyte character, -1 is returned. Otherwise, the return value is the number of bytes in the resulting multibyte character.

Since `wctomb` might be called repeatedly to produce successive multibyte characters in a string, it must remember the current shift state.

6.4 The `mbstowcs` Function

`mbstowcs` converts a given multibyte character string to an array of corresponding wide characters. For example:

```
#include <stdlib.h>
#include <stdio.h>

main()
{
        size_t i, j;
        wchar_t wc[5] = {'?', '?', '?', '?', '?'};
```

```
        printf("MB_CUR_MAX = %lu\n", (unsigned long)MB_CUR_MAX);

/*1*/   i = mbstowcs(wc, "AB", 5);
        printf("i = %lu, wc = ", (unsigned long)i);

        for (j = 0; j < 5; ++j) {
                printf(" %#4lX", (unsigned long)wc[j]);
        }
        putchar('\n');

/*2*/   i = mbstowcs(wc, "ABCDE", 5);
        printf("i = %lu, wc = ", (unsigned long)i);

        for (j = 0; j < 5; ++j) {
                printf(" %#4lX", (unsigned long)wc[j]);
        }
        putchar('\n');

        return 0;
}
```

The output produced on one ASCII-based system was:

```
MB_CUR_MAX = 1
i = 2, wc =  0X41 0X42    0 0X3F 0X3F
i = 5, wc =  0X41 0X42 0X43 0X44 0X45
```

In case 1, the string contains only two multibyte characters yet five were promised. As a result, these two are converted to wide characters and the trailing null character is also converted. In case 2, the string contains five multibyte characters, all of which are converted. However, the trailing null is not processed.

The multibyte string is assumed to begin in its initial shift state. If an invalid multibyte character is encountered, the value `(size_t)-1` is returned, otherwise the number of multibyte characters converted (not including any trailing null) is returned.

6.5 The `wcstombs` Function

`wcstombs` converts a given array of wide characters to a corresponding multibyte character string. For example:

```
#include <stdlib.h>
#include <stdio.h>

main()
{
        size_t i, j;
        wchar_t wcs1[5] = L"AB";
        wchar_t wcs2[] = L"ABCDEF";
        char string[] = {'?','?','?','?','?'};

        printf("MB_CUR_MAX = %lu\n", (unsigned long)MB_CUR_MAX);
```

```
/*1*/   i = wcstombs(string, wcs1, 5);
        printf("i = %lu, string = ", (unsigned long)i);

        for (j = 0; j < 5; ++j) {
                printf(" %#4lX", (unsigned long)string[j]);
        }
        putchar('\n');

/*2*/   i = wcstombs(string, wcs2, 5);
        printf("i = %lu, string = ", (unsigned long)i);

        for (j = 0; j < 5; ++j) {
                printf(" %#4lX", (unsigned long)string[j]);
        }
        putchar('\n');

        return 0;
}
```

The output produced on one ASCII-based system was:

```
MB_CUR_MAX = 1
i = 2, string =  0X41 0X42    0 0X3F 0X3F
i = 5, string =  0X41 0X42 0X43 0X44 0X45
```

In case 1, the array has only two wide characters which is less than expected. As a result, these two are converted to multibyte characters and the trailing null character is also converted. In case 2, the wide string contains more characters than expected, so only five bytes-worth of multibyte characters string is produced and no trailing null is appended.

The multibyte string produced begins in its initial shift state. If a wide character is encountered and it has no corresponding multibyte character, the value `(size_t)-1` is returned, otherwise the number of bytes written to the multibyte string (not including any trailing null) is returned.

7 ISO C Amendment 1

This amendment resulted in the creation of several new headers and some additions to existing ones, as follows:

- `<errno.h>` had the macro `EILSEQ` added to it.
- `<iso646.h>` was created as a repository for a family of macros that expand to alternate spellings for source characters necessary for C programming yet missing from the ISO 646 character set.
- `<wchar.h>` contains some type synonyms, a tag, some macros, and many functions, which provide for extended multibyte and wide-character processing.
- `<wctype.h>` contains some type synonyms, a macro, and many functions, which provide for wide-character classification and mapping.

Many of the new functions have names with spellings that parallel their single-byte counterparts. The complete list of new names and their parent headers is shown in the following table:

Amendment 1 Identifiers in Alphabetic Order					
Identifier	*Header*	*Identifier*	*Header*	*Identifier*	*Header*
and	iso646.h	mbstate_t	wchar.h	wcsncmp	wchar.
and_eq	iso646.h	not	iso646.h	wcsncpy	wchar.h
bitand	iso646.h	not_eq	iso646.h	wcspbrk	wchar.h
bitor	iso646.h	NULL	wchar.h	wcsrchr	wchar.h
btowc	wchar.h	or	iso646.h	wcsrtombs	wchar.h
compl	iso646.h	or_eq	iso646.h	wcsspn	wchar.h
EILSEQ	errno.h	putwc	wchar.h	wcsstr	wchar.h
fgetwc	wchar.h	putwchar	wchar.h	wcstod	wchar.h
fgetws	wchar.h	size_t	wchar.h	wcstok	wchar.h
fputwc	wchar.h	swprintf	wchar.h	wcstol	wchar.h
fputws	wchar.h	swscanf	wchar.h	wcstoul	wchar.h
fwide	wchar.h	tm	wchar.h	wcsxfrm	wchar.h
fwprintf	wchar.h	towctrans	wctype.h	wctob	wchar.h
fwscanf	wchar.h	towlower	wctype.h	wctrans	wctype.h
getwc	wchar.h	towupper	wctype.h	wctrans_t	wctype h
getwchar	wchar.h	ungetwc	wchar.h	wctype	wctype.h
iswalnum	wctype.h	vfwprintf	wchar.h	wctype_t	wctype.h
iswalpha	wctype.h	vswprintf	wchar.h	WEOF	wchar.h
iswcntrl	wctype.h	vwprintf	wchar.h	WEOF	wctype.h
iswctype	wctype.h	WCHAR_MAX	wchar.h	wint_t	wchar.
iswdigit	wctype.h	WCHAR_MIN	wchar.h	wint_t	wctype h
iswgraph	wctype.h	wchar_t	wchar.h	wmemchr	wchar.h
iswlower	wctype.h	wcrtomb	wchar.h	wmemcmp	wchar.h
iswprint	wctype.h	wcscat	wchar.h	wmemcpy	wchar.h
iswpunct	wctype.h	wcschr	wchar.h	wmemmove	wchar.h
iswspace	wctype.h	wcscmp	wchar.h	wmemset	wchar.h
iswupper	wctype.h	wcscoll	wchar.h	wprintf	wchar.h
iswxdigit	wctype.h	wcscpy	wchar.h	wscanf	wchar.h
mbrlen	wchar.h	wcscspn	wchar.h	xor	iso646.h
mbrtowc	wchar.h	wcsftime	wchar.h	xor_eq	iso646.h
mbsinit	wchar.h	wcslen	wchar.h		
mbsrtowcs	wchar.h	wcsncat	wchar.h		

Many of the functions defined in Amendment 1 are discussed in *Madell, Clark, and Abegg* listed in the attached reading list.

8 ISO 646 Issues

The ISO 646 character set does not include all of the characters needed for writing C source programs. In order to allow users of this character set to write C source, the C standards committee invented trigraphs, the header <iso646.h>, and digraphs.

None of these solutions can be promoted as good language design; they simply are the result of political compromise between the member nations involved. The debate over the need for, and contents of, the header <iso646.h> and digraphs lasted more than three years.

8.1 Trigraphs

A *trigraph* is an alternate spelling for a punctuation character needed in writing C source, but which is missing from national variants of the ISO 646 character set used by various machines and terminals in Europe. Trigraphs were designed to enable machine conversion of source.

A trigraph has the general form ??x where x is a punctuation character. The complete set of trigraphs is as follows:

Trigraphs	
Sequence	*Meaning*
??!	|
??'	^
??(	[
??)	]
??-	~
??/	\
??<	{
??=	#
??>	}

Since the compiler recognizes trigraphs before it tokenizes source a new escape sequence, \?, was invented to allow string literals to contain literal text of the form ??x without that text being interpreted as a trigraph.

The following example contains all the defined trigraphs:

```
/* example containing trigraphs */

??=define CCCC ??/
100

void test1()
??<
        char c??(10??)??(5??);
        int i = 10;

        i = ??-(4 ??' 10 ??! 3);
??>
```

The equivalent code without trigraphs is:

```
/* same example without trigraphs */

#define CCCC \
100

void test1()
{
        char c[10][5];
        int i = 10;

        i = ~(4 ^ 10 | 3);
}
```

8.2 <iso646.h>

It was *not* intended that programmers write source using trigraphs directly although that certainly is possible. Therefore, it was decided that Amendment 1 should contain a more-readable set of tokens for many of the trigraphs. This resulted in the addition of a series of macros defined in the new header <iso646.h>. The macros and their meanings are as follows:

<iso646.h> Contents	
Macro	*Equivalent Token*
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

Note: In the draft C++ Standard, these names are actually reserved words, *not* macros. The header <iso646.h> still exists, however, so existing C code that includes it will continue to compile under C++.

8.3 Digraphs

While the macros provided by <iso646.h> help source code readability, it was felt that more support was needed. And so the idea of digraphs was born. A *digraph* is an alternate spelling for another token. The following digraphs are defined in Amendment 1:

Digraphs	
Spelling	*Equivalent Token*
<:	[
:>	]
<%	{
%>	}
%:	#
%:%:	##

9 Recommended Reading

The following books and publications that will likely be of use to C programmers interested in further information on internationalization:

- *ANSI/ISO 9899:1990 Programming Language C.* American National Standards Institute, New York, NY. 1990, 219 pp.
- *ANSI/ISO 9899:1990 Programming Language C: Amendment 1.* American National Standards Institute, New York, NY. 1994, 52 pp.
- *Digital Guide to Developing International Software.* Digital Press, Bedford, MA. 1991, 381 pp., ISBN 55558-063-7.
- *ISO/IEC 9945-1. Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface.* American National Standards Institute, New York, NY.
- *ISO/IEC 9945-2. Portable Operating System Interface (POSIX) – Part 2: Shell and Utilities.* American National Standards Institute, New York, NY.
- *ISO/IEC 10646-1:1993 Universal Multiple-Octet Coded Character set (UCS) Part 1: Architecture and Basic Multilingual Plane.*
- Lunde, Ken. *Understanding Japanese Information Processing.* O'Reilly & Associates, Sebastopol, CA. 1993, 435 pp., ISBN 1-56592-043-0.

- Madell, Tom, Clark Parsons, and John Abegg. *Developing and Localizing International Software.* Hewlett-Packard Professional Books/Prentice Hall, Englewood Cliffs, NJ. 1994, 150 pp., ISBN 0-13-300674-3.
- Plauger, P.J., regular column in *C/C++ Users Journal.* R&D Publications, Lawrence, KS. 1990–1994.
- Plauger, P.J., regular column in *The Journal of C Language Translation.* IECC, Cambridge, MA. 1990–1994, ISSN 1042-5721.
- The Unicode Consortium. *The Unicode Standard: Worldwide Character Encoding Version 1.0, Volume 1.* Addison-Wesley, Reading, MA. 1990, 682 pp., ISBN 0-201-56788-1.
- The Unicode Consortium. *The Unicode Standard: Worldwide Character Encoding Version 1.0, Volume 2.* Addison-Wesley, Reading, MA. 1992, 439 pp., ISBN 0-201-60845-6.
- The Unicode Consortium. *The Unicode Standard, Version 1.1.* Changes from Version 1 in draft form from The Unicode Consortium. 1994.

Index