The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

5.2.1.2 Multibyte characters

multibyte character source contain

The source character set may contain multibyte characters, used to represent members of the extended character set. 238

Commentary

multibyte character translation phase 1
UCN models of

The mapping from physical source file multibyte characters to the source character set occurs in translation phase 1. Whether multibyte characters are mapped to UCNs, single characters (if possible), or remain as multibyte characters depends on the model used by the implementation.

C++

The representations used for multibyte characters, in source code, invariably involve at least one character that is not in the basic source character set:

2.1p1 *Any source file character not in the basic source character set (2.2) is replaced by the universal-character-name that designates that character.*

The C++ Standard does not discuss the issue of a translator having to process multibyte characters during translation. However, implementations may choose to replace such characters with a corresponding universal-character-name.

Other Languages

Most programming languages do not contain the concept of multibyte characters.

Common Implementations

universal character name syntax

Support for multibyte characters in identifiers, using a shift state encoding, is sometimes seen as an extension. Support for multibyte characters in this context using UCNs is new in C99. The most common implementations have been created to support the various Japanese character sets.

Coding Guidelines

The standard does not define how multibyte characters are to be represented. Any program that contains them is dependent on a particular implementation to do the right thing. Converting programs that existed before support for universal character names became available may not be economically viable.

Some coding guideline documents recommend against the use of characters that are not specified in the C Standard. Simply prohibiting multibyte characters because they rely on implementation-defined behavior ignores the cost/benefit issues applicable to the developers who need to read the source. These are complex issues for which your author has insufficient experience with which to frame any applicable guideline recommendations.

The execution character set may also contain multibyte characters, which need not have the same encoding as for the source character set. 239

Commentary

Multibyte characters could be read from a file during program execution, or even created by assigning byte values to contiguous array elements. These multibyte sequences could then be interpreted by various library functions as representing certain (wide) characters.

The execution character set need not be fixed at translation time. A program's locale can be changed at execution time (by a call to the `setlocale` function). Such a change of locale can alter how multibyte characters are interpreted by a library function.

C++

There is no explicit statement about such behavior being permitted in the C++ Standard. The C header **<wchar.h>** (specified in Amendment 1 to C90) is included by reference and so the support it defines for multibyte characters needs to be provided by C++ implementations.

Other Languages

Most languages do not include library functions for handling multibyte characters.

Coding Guidelines

Use of multibyte characters during program execution is an applications issue that is outside the scope of these coding guidelines.

---

240 For both character sets, the following shall hold:

Commentary

This is a set of requirements that applies to an implementation. It is the minimum set of guaranteed requirements that a program can rely on.

Coding Guidelines

The set of requirements listed in the following C-sentences is fairly general. Dealing with implementations that do not meet the requirements listed in these sentences is outside the scope of these coding guidelines.

---

241 — The basic character set shall be present and each character shall be encoded as a single byte.

Commentary

This is a requirement on the implementation. It prevents an implementation from being purely multibyte-based. The members of the basic character set are guaranteed to always be available and fit in a byte.

basic character set
fit in a byte

Common Implementations

An implementation that includes support for an extended character set might choose to define `CHAR_BIT` to be 16 (most of the commonly used characters in ISO 10646 are representable in 16 bits, each in UTF-16; at least those likely to be encountered outside of academic research and the traditional Chinese written on Hong Kong). Alternatively, an implementation may use an encoding where the members of the basic character set are representable in a byte, but some members of the extended character set require more than one byte for their encoding. One such representation is UTF-8.

extended character set
CHAR_BIT
macro
ISO 10646
UTF-16
UTF-8

---

242 — The presence, meaning, and representation of any additional members is locale-specific.

Commentary

On program startup the execution locale is the `"C"` locale. During execution it can be set under program control. The standard is silent on what the translation time locale might be.

Common Implementations

The full Ascii character set is used by a large number of implementations.

Coding Guidelines

It often comes as a surprise to developers to learn what characters the C Standard does not require to be provided by an implementation. Source code readability could be affected if any of these additional members appear within comments and cannot be meaningfully displayed. Balancing the benefits of using additional members against the likelihood of not being able to display them is a management issue.

The use of any additional members during the execution of a program will be driven by the user requirements of the application. This issue is outside the scope of these coding guidelines.

---

243 — A multibyte character set may have a *state-dependent encoding*, wherein each sequence of multibyte characters begins in an *initial shift state* and enters other locale-specific *shift states* when specific multibyte characters are encountered in the sequence.

multibyte character
state-dependent encoding
shift state

Commentary

State-dependent encodings are essentially finite state machines. When a state encoding, or any multibyte encoding, is being used the number of characters in a string literal is not the same as the number of bytes encountered before the null character. There is no requirement that the sequence of shift states and characters representing an extended character be unique.

extended characters

combining characters

There are situations where the visual appearance of two or more characters is considered to be a single character. For instance, (using ISO 10646 as the example encoding), the two characters *LATIN SMALL LETTER O* (U+006F) followed by *COMBINING CIRCUMFLEX ACCENT* (U+0302) represent the grapheme cluster (the ISO 10646 term[1] for what might be considered a *user character*) **ô** not the two characters **o ^**. Some languages use grapheme clusters that require more than one combining character, for instance ô̲. Unicode (not ISO 10646) defines a canonical accent ordering to handle sequences of these combining characters. The so-called *combining characters* are defined to combine with the character that comes immediately before them in the character stream. For backwards compatibility with other character encodings, and ease of conversion, the ISO 10646 Standard provides explicit codes for some accent characters; for instance, *LATIN SMALL LETTER O WITH CIRCUMFLEX* (U+00F4) also denotes **ô**.

A character that is capable of standing alone, the **o** above, is known as a *base* character. A character that modifies a base character, the **ô** above, is known as a *combining* character (the visible form of some combining characters are called *diacritic* characters). Most character encodings do not contain any combining characters, and those that do contain them rarely specify whether they should occur before or after the modified base character. Claims that a particular standard require the combining character to occur before the base character it modifies may be based on a misunderstanding. For instance, ISO/IEC 6937 specifies a single-byte encoding for base characters and a double-byte encoding for some visual combinations of (diacritic + base) Latin letter. These double-byte encodings are precomposed in the sense that they represent a single character; there is no single-byte encoding for the diacritic character, and the representation of the second byte happens to be the same as that of the single-byte representation of the corresponding base character (e.g., 0xC14F represents *LATIN CAPITAL LETTER O WITH GRAVE* and 0xC16F represents *LATIN SMALL LETTER O WITH GRAVE*).

C90

The C90 Standard specified implementation-defined shift states rather than locale-specific shift states.

C++

The definition of multibyte character, 1.3.8, says nothing about encoding issues (other than that more than one byte may be used). The definition of multibyte strings, 17.3.2.1.3.2, requires the multibyte characters to begin and end in the initial shift state.

Common Implementations

ISO 2022

Most methods for state-dependent encoding are based on ISO/IEC 2022:1994 (identical to the standard ECMA-35 “Character Code Structure and Extension Techniques”, freely available from their Web site, http://www.ecma.ch). This uses a different structure than that specified in ISO/IEC 10646–1. The encoding method defined by ISO 2022 supports both 7-bit and 8-bit codes. It divides these codes up into control characters (known as *C0* and *C1*) and graphics characters (known as *G0*, *G1*, *G2*, and *G3*). In the initial shift state the *C0* and *G0* characters are in effect.

Table 243.1: Commonly seen ISO 2022 Control Characters. The alternative values for SS2 and SS3 are only available for 8-bit codes.

Name	Acronym	Code Value	Meaning
Escape	ESC	0x1b	Escape
Shift-In	SI	0x0f	Shift to the G0 set
Shift-Out	SO	0x0e	Shift to the G1 set
Locking-Shift 2	LS2	ESC 0x6e	Shift to the G2 set
Locking-Shift 3	LS3	ESC 0x6f	Shift to the G3 set
Single-Shift 2	SS2	ESC 0x4e, or 0x8e	Next character only is in G2
Single-Shift 3	SS3	ESC 0x4f, or 0x8f	Next character only is in G3

Some of the control codes and their values are listed in Table 243.1. The codes SI, SO, LS2, and LS3 are known as *locking shifts*. They cause a change of state that lasts until the next control code is encountered. A stream that uses locking shifts is said to use *stateful encoding*.

ISO 2022 specifies an encoding method: it does not specify what the values within the range used for graphic characters represent. This role is filled by other standards, such as ISO 8859. A C implementation that supports a state-dependent encoding chooses which character sets are available in each state that it supports (the C Standard only defines the character set for the initial shift state). ISO 8859

Table 243.2: An implementation where G1 is ISO 8859–1, and G2 is ISO 8891–7 (Greek).

Encoded values	0x62	0x63	0x64	0x0e	0xe6	0x1b	0x6e	0xe1	0xe2	0xe3	0x0f
Control character				SO		LS2					SI
Graphic character	a	b	c		æ			α	β	γ	

Having to rely on implicit knowledge of what character set is intended to be used for G1, G2, and so on, is not always satisfactory. A method of specifying the character sets in the sequence of bytes is needed. The ESC control code provides this functionality by using two or more following bytes to specify the character set (ISO maintains a *registry of coded character sets*). It is possible to change between character sets without any intervening characters. Table 243.3 lists some of the commonly used Japanese character sets.

C source code written by Japanese developers probably has the highest usage of shift sequences. There are several JIS (Japanese Industrial Standard) documents specifying representations for such sequences. Shift JIS (developed by Microsoft) belies its name and does not involve shift sequences that use a state-dependent encoding.

Table 243.3: ESC codes for some of the character sets used in Japanese.

Character Set	Byte Encoding	Visible Ascii Representation
JIS C 6226–1978	1B 24 40	<ESC> $ @
JIS X 0208–1983	1B 24 42	<ESC> $ B
JIS X 0208–1990	1B 26 40 1B 24 42	<ESC> & @ <ESC> $ B
JIS X 0212–1990	1B 24 28 44	<ESC> $ (D
JIS-Roman	1B 28 4A	<ESC> (J
Ascii	1B 28 42	<ESC> (B
Half width Katakana	1B 28 49	<ESC> (I

Table 243.4: A JIS encoding of the character sequence かな漢字(“kana and kanji”).

Encoded values	0x1b	0x24	0x42	0x242b	0x244a	0x3441	0x3b7a	0x1b	0x28	0x4a
Control character	<ESC>	$	B					<ESC>	(	J
Graphic character				か	な	漢	字			
Ascii characters				$+	$J	4A	;z			

Coding Guidelines

Developers do not need to remember the numerical values for extended characters. The editor, or program development environment, used to create the source code invariably looks after the details (generating any escape sequences and the appropriate byte values for the extended character selected by the developer). How these tools decide to encode multibyte character sequences is outside the scope of these coding guidelines.

It is usually possible to express an extended character in a minimal number of bytes using a particular state-dependent encoding. The extent to which developers might create fixed-length data structures on the assumption that multibyte characters will not contain any redundant shift sequences is outside the scope of this book. The value of the MB_LEN_MAX macro places an upper limit on the number of possible redundant shift sequences.

footnote 152
MB_LEN_MAX

Example

```
#include <stdio.h>

char *p1 = "^[$B$3$l$OF|K\8lI=8=^[(J";  /* ^[$BF|K\8lJ8;zNs^[(J */
char *p2 = "^[$B$3$l$OF|1Q^[(Jmixed^[$BJ8;zNs^[(J";     /* Ascii + ^[$BF|K\8l^[(J */
char *p3 = "^[$B$3$l$OH>3Q^[(J^N6@6E^O^[$B$H^[(JASCII^[$B:.9g^[(J";

int main(void)
{
printf("%s^[$B$H^[(J%s^[$B$H^[(J%s\n", p1, p2, p3);
}
```

While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. 244

Commentary

The implementation of a stateful encoding has to pick a special character, which is not in the basic character set, to indicate the start of a shift sequence. When not in the initial shift state, it is very unlikely that single bytes will be interpreted the same way as when in the initial shift state.

C++

The C++ Standard does not explicitly specify this requirement.

Common Implementations

The ESC character, 0x1b, is commonly used to indicate the start of a shift sequence.

---

12) The trigraph sequences enable the input of characters that are not defined in the Invariant Code Set as described in ISO/IEC 646, which is a subset of the seven-bit US ASCII code set. 245

footnote 12

Commentary

When trigraphs are used, it is possible to write C source code that contains only those characters that are in the Invariant Code Set of ISO/IEC 646.

C90

The C90 Standard explicitly referred to the 1983 version of ISO/IEC 646 standard.

The interpretation for subsequent bytes in the sequence is a function of the current shift state. 246

Commentary

This wording is really a suggestion for the design of multibyte shift states (it is effectively describing the processing performed by finite state machines, which is what a shift state encoding is). Being able to interpret

a byte independent of the current shift state would indicate that the sequence of bytes that resulted in the current state were redundant.

The specification of the macro `MB_LEN_MAX` requires that the maximum number of bytes needed to handle a supported multibyte character be provided. It may, or may not, be possible to represent some redundant shift sequence within the available bytes. The standard does not explicitly require or prohibit support for redundant shift sequences. MB_LEN_MAX

C++

A set of virtual functions for handling state-dependent encodings, during program execution, is discussed in Clause 22, Localization library. But, this requirement is not specified.

Common Implementations

Implementations usually use a simple finite state machine, often automatically generated, to handle the mapping of shift states into their execution character value. The extent to which sequences of redundant shift sequences is supported will depend on the implementation.

Coding Guidelines

The sequence of bytes in a shift sequence are usually generated via some automated process. For this reason a guideline recommending against the use of redundant shift sequences is unlikely to be enforceable, and none is given.

---

247 — A byte with all bits zero shall be interpreted as a null character independent of shift state. byte all bits zero

Commentary

This is a requirement on the implementation. This requirement makes it possible to search for the end of a string without needing any knowledge of the encoding that has been used. For instance, string-handling functions can copy multibyte characters without interpreting their contents.

C++

. . . , plus a null character (respectively, null wide character), whose representation has all zero bits. 2.2p3

While the C++ Standard does not rule out the possibility of all bits zero having another interpretation in other contexts, other requirements (17.3.2.1.3.1p1 and 17.3.2.1.3.2p1) restrict these other contexts, as do existing character set encodings.

---

248 — ~~A byte with all bits zero shall not occur in the second or subsequent bytes of a~~ Such a byte shall not occur as part of any other multibyte character. multibyte character end in initial shift state

Commentary

This is a requirement on the implementation. The effect of this requirement is that partial multibyte characters cannot be created (otherwise the behavior is undefined). A null character can only exist outside of the sequence of bytes making up a multibyte character. For source files this requirement follows from the requirement to end in the initial shift state. During program execution this requirement means that library functions processing multibyte characters do not need to concern themselves with handling partial multibyte characters at the end of a string. 250 token shift state

The wording was changed by the response to DR #278 (it is a requirement on the implementation that forbids a two-byte character from having a first, or any, byte that is zero).

C++

This requirement can be deduced from the definition of null terminated byte strings, 17.3.2.1.3.1p1, and null terminated multibyte strings, 17.3.2.1.3.2p1.

For source files, the following shall hold: 249

Commentary

These C-sentences specify requirements on a program. A program that violates them exhibits undefined behavior.

locale-specific behavior
implementation-defined behavior

Use of multibyte characters can involve locale-specific and implementation-defined behaviors. A source file does not affect the conformance status of any program built using it, provided its use of multibyte characters either involves locale-specific behavior or the implementation-defined behavior does not affect program output (e.g., they appear in comments).

Coding Guidelines

The creation of multibyte characters within source files is usually handled by an editor. The developer involvement in the process being the selection of the appropriate character. In such an environment the developer has no control over the byte sequences used. A guideline recommending against such usage is likely to be impractical to implement and none is given.

token
shift state

— An identifier, comment, string literal, character constant, or header name shall begin and end in the initial shift state. 250

Commentary

These are the only tokens that can meaningfully contain a multibyte character. A token containing a multibyte character should not affect the processing of subsequent tokens. Without this requirement a token that did not end in the initial shift state would be likely to affect the processing of subsequent tokens.

C90

Support for multibyte characters in identifiers is new in C99.

C++

translation phase 1

In C++ all characters are mapped to the source character set in translation phase 1. Any shift state encoding will not exist after translation phase 1, so the C requirement is not applicable to C++ source files.

Coding Guidelines

The fact that many multibyte sequences are created automatically, by an editor, can make it very difficult for a developer to meet this requirement. A developer is unlikely to intentionally end a preprocessing token, created using a multibyte sequence, in other than the initial state. A coding guideline is unlikely to be of benefit.

— An identifier, comment, string literal, character constant, or header name shall consist of a sequence of valid multibyte characters. 251

Commentary

What is a valid multibyte character? This decision can only be made by a translator, should it chose to accept multibyte characters.

In C90 it was relatively easy to lexically process a source file containing multibyte characters. The context in which these characters occurred often meant that a lexer simply had to look for the character that terminated the kind of token being processed (unless that character occurred as part of a multibyte character).

Identifier tokens do not have a single termination character. This means that it is not possible to generalise support for multibyte characters in identifiers across all translators. It is possible that source containing a multibyte character identifier supported by one translator will cause another translator to issue a diagnostic.

C90

Support for multibyte characters in identifiers is new in C99.

C++

In C++ all characters are mapped to the source character set in translation phase 1. Any shift state encoding will not exist after translation phase 1, so the C requirement is not applicable to C++ source files.

translation phase 1

Coding Guidelines

In some cases source files can contain multibyte characters and be translated by translators that have no knowledge of the structure of these multibyte characters. The developer is relying on the translator ignoring them in comments containing their native language, or simply copying the character sequence in a string literal into the program image. In other cases, for instance identifiers, knowledge of the encoding used for the multibyte character set is likely to be needed by a translator.

Ensuring that a translator capable of handling any multibyte characters occurring in the source is used, is a configuration-management issue that is outside the scope of these coding guidelines.

References

1. M. Davis. Text boundaries. Technical Report 29, The Unicode Consortium, Mar. 2005.