The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

5.2.1 Character sets

source character set
execution character set

Two sets of characters and their associated collating sequences shall be defined: the set in which source files are written (the *source character set*), and the set interpreted in the execution environment (the *execution character set*). 214

Commentary

This is a requirement on the implementation. This defines the terms *source character set* and *execution character set*.

A collating sequence is defined, but the particular values for the characters are not specified. These, and only these, characters are guaranteed to be supported by a conforming implementation.

The purpose for defining these two character sets is to separate out the environment in which the source is translated from the environment in which the translated output is executed. When these two environments are different, the translation process is commonly known as *cross compiling*. An implementation may add additional characters to either the source or execution character set. There is no requirement that any additional characters exist in either environment.

The characters used in the definition of the C language exist within both the source and execution character sets. It is intended that a C translator be able to successfully translate a C translator written in C.

Rationale

ISO (the International Organization for Standardization) uses three technical terms to describe character sets: repertoire, collating sequence, and codeset. The repertoire is the set of distinct printable characters. The term abstracts the notion of printable character from any particular representation; the glyphs R, R, R, *R*, *R*, R, and **R**, all represent the same element of the repertoire, "upper-case-R", which is distinct from "lower-case-r". Having decided on the repertoire to be used (C needs a repertoire of 91 characters plus whitespace), one can then pick a collating sequence which corresponds to the internal representation in a computer. The repertoire and collating sequence together form the codeset.

What is needed for C is to determine the necessary repertoire, ignore the collating sequence altogether (it is of no importance to the language), and then find ways of expressing the repertoire in a way that should give no problems with currently popular codesets.

C90

The C90 Standard did not explicitly define the terms *source character set* and *execution character set*.

C++

The C++ Standard does not contain a requirement to define a collating sequence on the character sets it specifies.

Other Languages

Many languages are designed for a hosted environment and do not need to make the distinction between source and execution character sets. Ada is explicitly defined in terms of the Ascii character set. After all, it was designed as a language for use by the US Department of Defense.

Common Implementations

On most implementations the two characters sets have the same representations.

Coding Guidelines

Developers whose native tongue is English tend to be unaware of the distinction between source and execution character sets. Most of these developers do most of their development in environments where they are identical.

basic character set
extended characters

Each set is further divided into a *basic character set*, whose contents are given by this subclause, and a set of zero or more locale-specific members (which are not members of the basic character set) called *extended characters*. 215

Commentary

This defines the terms *basic character set* and *extended characters*. It separates out the two components of the character set used by an implementation; the one which is always required to be provided and the extended set which is optional.

C90

This explicit subdivision of characters into sets is new in C99. The wording in the C90 Standard specified the minimum contents of the basic source and basic execution character sets. These terms are now defined exactly, with all other characters being called extended characters.

214 source character set

> *. . . ; any additional members beyond those required by this subclause are locale-specific.*

C++

> *The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.* 2.2p3

The C++ Standard more closely follows the C90 wording.

Other Languages

In the past most programming languages tended not say anything about supporting other character set members, although most implementations usually supported some additional characters. This situation is starting to change as the predominantly English-speaking standards world starts to recognize the importance of supporting programs written in the developer's native character set (the introduction of ISO 10646 also helped).

ISO 10646

Common Implementations

A large number of C translators originate in the USA, or target this market. This locale usually requires support for extended characters in the form of those members in the Ascii character set that are not in the basic source character set (the *$* and *@* characters being the most obvious).

Vendors selling into non-English-speaking markets commonly add support for extended characters in the execution character set to support a native language. These implementations usually also support the occurrence of extended characters in comments. Support for extended characters outside of character constants, string literals, and comments has been much less common.

Coding Guidelines

Using extended characters to make applications comprehensible to their users is obviously essential. However, the handling of such characters is part of the application domain and outside the scope of these coding guidelines. The issues involved in programs written in one locale targeted at another locale is also largely outside the scope of these coding guidelines.

Using characters from the developer's native language can make an important contribution to program readability for those developers who share that native language. Some applications are now developed using people whose native languages differ from each other. The issue of using extended characters to improve source code readability is not always clear-cut. What is the best way to handle programs made up of translation units developed by developers from different locales; should all developers working on the same application use the same locale? To a large extent these questions involve predicting the future. Who will be doing the future development and maintenance of the software? It may not be possible to provide a reliable answer to this question. The issue of what characters to use in identifier names is discussed elsewhere.

identifier syntax

Example

```
1   char dollar = '$';
```

extended character set

The combined set is also called the *extended character set*. 216

Commentary

A somewhat confusing use of terminology. A developer might be forgiven for thinking that this term applied to the set of extended characters only. For both the source character set and the execution character set, the following statement is true:

```
1   extended_character_set = basic_character_set + extended_characters;
```

Coding Guidelines

A coding guideline document needs to be very careful in its use of terminology when dealing with character set issues.

The values of the members of the execution character set are implementation-defined. 217

Commentary

translation phase 1

basic character set 222 fit in a byte

This has already been stated for the members of the source character set. Although it might not specify their values, the standard does specify some of the properties of objects that hold them.

Common Implementations

Many implementations use the Ascii character set, with the EBCDIC character set appearing to be restricted to use on IBM mainframes and their derivatives. Most implementations use the same values for the corresponding members of both the source and execution character set.

Coding Guidelines

Developers tend to make several assumptions about the values of the execution character set:

- They are the same as the source character set.
- All the uppercase letters, all the lowercase letters, and all the digits are contiguous.
- They are less than 128.
- The actual values used by a translator (e.g., space being 32).

digit characters 223 contiguous

Only the assumption about the digits being contiguous is guaranteed to be true.

A program may contain implicit dependencies on the representation of members of the execution character set because developers are not aware they are making assumptions about something that is not fixed. Designing programs to accommodate the properties of different character sets is not a trivial matter that can be covered in a few guideline recommendations.

Example

```
1   #if 'a' == 99 /* Not the execution character set. */
2   #endif
3
4   int f(void)
5   {
6   return 'b' == 88;
7   }
```

---

execution character set represented by

In a character constant or string literal, members of the execution character set shall be represented by corresponding members of the source character set or by *escape sequences* consisting of the backslash \ followed by one or more characters. 218

Commentary

This describes the two methods specified by the standard for representing members of the execution character set in character constants and string literals. They are one route through which characters appearing in the source code can appear in the output produced by a program. Comments are removed in translation phase 3. Identifier spellings are represented by objects in the program image, and are not directly involved in execution-time behavior.

comment
replaced by space

Escape sequences are a method of representing execution characters in the source, which may not be representable in the source character set. They make it possible to explicitly specify a particular numeric value, which is known to represent a given character in the execution character set (as defined by the implementation).

Other Languages

The convention of mapping source characters to their corresponding execution characters is common to the majority of programming languages. Some of the more recently designed, or updated, programming languages also support some form of escape sequence mechanism.

Common Implementations

The POSIX locale mechanism defines a representation for characters based on their names. For instance, *LETTER-A* is used to denote the character *A*. This approach removes the need for representing characters on keyboards and displays. The main use, to date, for this character specification methodology has been within POSIX locale specifications, where the meaning of a sequence of one or more characters might otherwise be uncertain.

Coding Guidelines

String literals are not always used to simply represent character sequences. A developer may choose to embed other, numeric, information within a string literal. Relying on characters to have the desired value would create a dependence on a particular character set being used and create literals that were harder to interpret (use of escape sequences makes it explicit that a numeric value is required). The contents of string literals therefore need to be interpreted in the context in which they are used.

The value of an escape sequence may, or may not, be the same as that of a member of the basic character set. The extent to which the value of an escape sequence does, or does not, represent a member of the basic character set is one of intent on the part of the developer. This issue is discussed elsewhere.

escape sequence
syntax

Example

```
1  #include <stdio.h>
2
3  void f(void)
4  {
5  printf("\110ello World\n"); /* ASCII \110 == H */
6  printf("Be \100 one\n");    /* @ symbol not on an ASCII keyboard. */
7  }
```

---

219 A byte with all bits set to 0, called the *null character*, shall exist in the basic execution character set;

null character

Commentary

This defines the term *null character* (an equivalent one for a null wide character is given in the library section). The null character is used to terminate string literals.

string literal
zero appended

The C committee once received a request from a communications-related standards committee asking that this requirement be removed from the C Standard. The sending of null bytes was causing problems on some communications links. The C committee pointed out that C's usage was a long-established practice and that they had no plans to change it.

Other Languages

Some form of null character occurs in any language that uses a terminating character (rather than a length count) to represent strings. SQL has a null value. It is used to indicate *value unknown*, or *value not present* (no two NULL values can ever compare equal).

Coding Guidelines

There is a common beginner's mistake that is sometimes not diagnosed because an implementation has defined the `NULL` macro to be `0`, rather than `(void *)0`. If C++ compatible headers are being used, the problem is not helped by that language's explicit requirement that the null pointer be represented by `0`.

---

character string terminate

it is used to terminate a character string. 220

Commentary

A string literal may contain more than one null character, or none at all. In the former case the literal will contain more than one string (according to the definition of that term given in the library section) — for instance, `"abc\000xyz"` and `char s[3] = "abc"`. Each null character terminates a string even though more than one of them may appear in a string literal.

C++

2.13.4p4 *After any necessary concatenation, in translation phase 7 (2.1), `'\0'` is appended to every string literal so that programs that scan a string can find its end.*

In practice the C usage is the same as that specified by C++.

Other Languages

Not all languages specify a particular representation for strings. Some implementations of these languages use a numeric count, usually stored before the first character of the string, representation to specify the length of the string. Many Pascal implementations use this approach.

In some languages the representation of strings is completely hidden from the developer. Perl and Snobol have sophisticated built-in support for string creation and manipulation. Their implementations can use whatever representation techniques they choose. Programs attempting to access the representation, for this kind of language, are failing to operate in the algorithmic domain that the language designers intended.

Common Implementations

The MrC[2] compiler from Apple Computer provides the escape sequence \p so that developers can specify that a string literal is a *Pascal string*. It has to occur as the first character and causes the implementation to maintain a byte count, rather than a null terminator.

Coding Guidelines

Null characters are different from other escape sequences in a string literal in that they have the special status of acting as a terminator (e.g., the library string searching routines terminate when a null character is encountered, leaving any subsequent characters in the literal unchecked). Any surprising behavior occurring because of this usage is a fault and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

guidelines not faults

---

basic source character set
basic execution character set

Both the basic source and basic execution character sets shall have the following members: the 26 *uppercase letters* of the Latin alphabet 221

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
```

the 26 *lowercase letters* of the Latin alphabet

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

the 10 decimal *digits*

```
0 1 2 3 4 5 6 7 8 9
```

the following 29 graphic characters

```
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
```

the space character, and control characters representing horizontal tab, vertical tab, and form feed.

Commentary

The original C Standard's work aimed to codify existing practice, and the K&R definition used the preceding collection of characters. The character values occupied by the #, [,], {, }, & and | characters in the Ascii character set are sometimes used to represent different characters in some Scandinavian character sets. Whether an awareness of this issue would have made any difference to the characters chosen can be debated (as could the extent to which a problem experienced by a minority of developers should have any noticeable impact on the majority of developers). It certainly made no difference to the design of Java, which occurred well after this issue had become well-known.

The characters vertical tab, form feed, carriage return, and new-line are sometimes referred to as line break characters. This term describes the most commonly seen visual effect of their appearance in a text file, not how a translator is required to interpret them.

C90

The C90 Standard referred to these characters as the *English alphabet*.

C++

The basic source character set consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphics characters: 2.2p1

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

The C++ Standard includes new-line in the basic source character set (C only includes it in the basic execution character set). 226 basic execution character set control characters

The C++ Standard does not separate out the uppercase, lowercase, and decimal digits from the graphical characters, so technically they are not defined for the basic source character set (the library functions such as `toupper` effectively define these terms for the execution character set).

Other Languages

Not all languages require as many characters to be supported as C does. The APL language contains so many characters especially designed for the language that they have their own standard – ISO 2575 *Registered Character Set 68— APL* – to describe them.

Common Implementations

The basic execution character set does not include three printable characters that appear in the first 127 positions of the ISO 10646 standard (and also in Ascii)— *$* (dollar), @ (commercial at), and ‘ (grave accent)

Coding Guidelines

A high priority might be given to supporting the *$* and @ characters in a money-oriented, wired world. But what about other characters that are not in the basic character set? When these characters are intended to appear in the source character set, the issue is one of displaying and potentially inputting them to a source file. Experience shows that developers usually have access to computers capable of displaying and inputting characters from the Ascii character set. This is an issue that is considered to be outside the scope of these coding guidelines. When these characters are intended to appear in the execution character set, it becomes an applications issue that is outside the scope of these coding guidelines.

Horizontal tab is a single white-space character. However, when viewing source code containing such a character, many display devices appear to replace it with more than one white-space character. There is no agreed-on spacing for the horizontal tab character and its use can cause the appearance of the source code to vary between display devices. The standard contains an alternative method of representing horizontal tab in string literals and character constants.

horizontal tab escape sequence

Most source code is displayed using a fixed-width font. Research[1] has shown that people read text faster (a 6% time difference, most of which can be attributed to the greater amount of information that a variable-width font allows to appear within the readers visual field) when it is displayed in a variable-width font than a fixed-width font. Comparing text has also been found to be quicker, but not searching for specific words. Use of a variable width font would also enable more characters to be displayed on a line, reducing the need to split statements across more than one line.

visual field

Support for use of variable-width fonts is not always available to developers. The issue of source, written using a variable-width font, having to be read in an environment where only a fixed-width font is available also needs to be considered (long lines may not be displayed as intended).[221.1]

[221.1] Note that source code displayed in this book uses a fixed-width font. This usage is intended to act as a visual aid in distinguishing code from text and not as an endorsement of fixed-width fonts.

Table 221.1: Occurrence of characters as a percentage of all characters and as a percentage of all noncomment characters (i.e., outside of comments). Based on the visible form of the `.c` files. For a comparison of letter usage in English language and identifiers see Figure **??**.

Letter or ASCII Value	All	Non-comment	Letter or ASCII Value	All	Non-comment	Letter or ASCII Value	All	Non-comment	Letter or ASCII Value	All	Non-comment
0	0.000	0.000	**sp**	15.083	13.927	@	0.009	0.002	‘	0.004	0.002
1	0.000	0.000	**!**	0.102	0.127	**A**	0.592	0.642	**a**	3.132	2.830
2	0.000	0.000	**"**	0.376	0.471	**B**	0.258	0.287	**b**	0.846	0.812
3	0.000	0.000	**#**	0.175	0.219	**C**	0.607	0.663	**c**	2.168	2.178
4	0.000	0.000	**$**	0.005	0.003	**D**	0.461	0.523	**d**	2.184	2.176
5	0.000	0.000	**%**	0.105	0.135	**E**	0.869	1.012	**e**	5.642	4.981
6	0.000	0.000	**&**	0.237	0.311	**F**	0.333	0.355	**f**	1.666	1.725
7	0.000	0.000	**’**	0.101	0.080	**G**	0.243	0.263	**g**	0.923	0.906
8	0.000	0.000	**(**	1.372	1.751	**H**	0.146	0.155	**h**	1.145	0.777
\t	3.350	4.116	**)**	1.373	1.751	**I**	0.619	0.643	**i**	3.639	3.469
\n	3.630	4.229	*****	1.769	0.769	**J**	0.024	0.026	**j**	0.074	0.077
11	0.000	0.000	**+**	0.182	0.233	**K**	0.098	0.116	**k**	0.464	0.481
12	0.003	0.004	**,**	1.565	1.914	**L**	0.528	0.609	**l**	2.033	1.915
\r	0.001	0.001	**-**	1.176	0.831	**M**	0.333	0.366	**m**	1.245	1.229
14	0.000	0.000	**.**	0.512	0.387	**N**	0.557	0.610	**n**	3.225	2.989
15	0.000	0.000	/	0.718	0.519	**O**	0.467	0.517	**o**	2.784	2.328
16	0.000	0.000	**0**	1.465	1.694	**P**	0.460	0.508	**p**	1.505	1.551
17	0.000	0.000	**1**	0.502	0.551	**Q**	0.033	0.037	**q**	0.121	0.135
18	0.000	0.000	**2**	0.352	0.408	**R**	0.652	0.729	**r**	3.405	3.254
19	0.000	0.000	**3**	0.227	0.262	**S**	0.691	0.758	**s**	3.166	2.961
20	0.000	0.000	**4**	0.177	0.203	**T**	0.686	0.740	**t**	4.566	4.200
21	0.000	0.000	**5**	0.149	0.171	**U**	0.315	0.349	**u**	1.575	1.510
22	0.000	0.000	**6**	0.176	0.209	**V**	0.128	0.149	**v**	0.662	0.682
23	0.000	0.000	**7**	0.131	0.144	**W**	0.131	0.135	**w**	0.494	0.385
24	0.000	0.000	**8**	0.184	0.207	**X**	0.213	0.254	**x**	0.870	1.002
25	0.000	0.000	**9**	0.128	0.122	**Y**	0.091	0.094	**y**	0.515	0.435
26	0.000	0.000	**:**	0.192	0.176	**Z**	0.027	0.033	**z**	0.125	0.135
27	0.000	0.000	**;**	1.276	1.670	**[**	0.163	0.210	**{**	0.303	0.401
28	0.000	0.000	**<**	0.118	0.147	\	0.097	0.126	**|**	0.098	0.124
29	0.000	0.000	**=**	1.039	1.042	**]**	0.163	0.210	**}**	0.303	0.401
30	0.000	0.000	**>**	0.587	0.762	**^**	0.003	0.002	~	0.009	0.012
31	0.000	0.000	**?**	0.022	0.019	**_**	2.550	3.238	**127**	0.000	0.000

Table 221.2: Relative frequency (most common to least common, with parenthesis used to bracket extremely rare letters) of letter usage in various human languages (the English ranking is based on the British National Corpus). Based on Kelk.[4]

Language	Letters
English	etaoinsrhldcumfpgwybvkxjqz
French	esaitnrulodcmpévqfbghjàxèyêâçîùôûïkëw
Norwegian	erntsilakodgmvfupbhøjyåæcwzx(q)
Swedish	eantrsildomkgväfhupåöbcyjxwzéq
Icelandic	anriestuðlgmkfhvoáþídjóbyæúöpéýcxwzq
Hungarian	eatlnskomzrigáéydbvhjőfupöócűíúüxw(q)

222 The representation of each member of the source and execution basic character sets shall fit in a byte.

basic character set fit in a byte

Commentary

This is a requirement on the implementation. The definition of character already specifies that it fits in a byte. However, a character constant has type **int**; which could be thought to imply that the value representation of

character single-byte
character constant type

basic character set positive if stored in char object

characters need not fit in a byte. This wording clarifies the situation. The representation of members of the basic execution character set is also required to be a nonnegative value.

C++

1.7p1 *A byte is at least large enough to contain any member of the basic execution character set and . . .*

This requirement reverses the dependency given in the C Standard, but the effect is the same.

Common Implementations

On hosts where characters have a width 16 or 32 bits, that choice has usually been made because of addressability issues (pointers only being able to point at storage on 16- or 32-bit address boundaries). It is not usually necessary to increase the size of a byte because of representational issues to do with the character set.

CHAR_BIT macro

In the EBCDIC character set, the value of `'a'` is 129 (in Ascii it is 97). If the implementation-defined value of `CHAR_BIT` is 8, then this character, and some others, will not be representable in the type **`signed char`** (in most implementations the representation actually used is the negative value whose least significant eight bits are the same as those of the corresponding bits in the positive value, in the character set). In such implementations the type **`char`** will need to have the same representation as the type **`unsigned char`**.

The ICL 1900 series used a 6-bit byte. Implementing this requirement on such a host would not have been possible.

representation information using ??

Coding Guidelines

A general principle of coding guidelines is to recommend against the use of representation information. In this case the standard is guaranteeing that a character will fit within a given amount of storage. Relying on this requirement might almost be regarded as essential in some cases.

Example

```
1  void f(void)
2  {
3  char C_1 = 'W';         /* Guaranteed to fit in a char. */
4  char C_2 = '$';         /* Not guaranteed to fit in a char. */
5  signed char C_3 = 'W'; /* Not guaranteed to fit in a signed char. */
6  }
```

---

digit characters contiguous

In both the source and execution basic character sets, the value of each character after **0** in the above list of decimal digits shall be one greater than the value of the previous. 223

Commentary

This is a requirement on the implementation. The Committee realized that a large number of existing programs depended on this statement being true. It is certainly true for the two major character sets used in the English-speaking world, Ascii, EBCDIC, and all of the human language digit encodings specified in Unicode, see Table **??**. The Committee thus saw fit to bless this usage.

Not only is it possible to perform relational comparisons on the digit characters (e.g, `'0'<'1'` is always true) but arithmetic operations can also be performed (e.g., `'0'+1 == '1'`). A similar statement for the alphabetic characters cannot be made because it would not be true for at least one character set in common use (e.g., EBCDIC).

C++

The above wording has been proposed as the response to C++ DR #173.

Other Languages

Most languages that have not recently had their specifications updated do not specify any representational properties for the values of their execution character sets. Java specifies the use of the Unicode character set (newer versions of the language specify newer versions of the Unicode Standard; all of which are the same as Ascii for their first 128 values), so this statement also holds true. Ada specifies the subset of ISO 10646 known as the Basic Multilingual Plane (the original language standard specified ISO 646). ISO 10646

Coding Guidelines

This requirement on an implementation provides a guarantee of representation information that developers can make use of (e.g., in relational comparisons, see Table **??**). The following are suggested wordings for deviations from the guideline recommendation dealing with making use of representation information. ?? representation information using

Dev ??

An integer character constant denoting a digit character may appear in the visible source as the operand of an *additive-operator*.

Example

```
1  #include <stdio.h>
2
3  extern char c_glob = '4';
4
5  int main(void)
6  {
7  if ('0' + 3 == '3')
8     printf("Sentence 221 is TRUE\n");
9
10 if (c_glob < '5')
11    printf("Sentence 221 may be TRUE\n");
12 if (c_glob < 53) /* '5' == 53 in ASCII */
13    printf("Sentence 221 does not apply\n");
14 }
```

---

224 In source files, there shall be some way of indicating the end of each line of text; end-of-line representation

Commentary

This is a requirement on the implementation.

The C library makes a distinction between text and binary files. However, there is no requirement that source files exist in either of these forms. The worst-case scenario: In a host environment that did not have a native method of delimiting lines, an implementation would have to provide/define its own convention and supply tools for editing such files. Some integrated development environments do define their own conventions for storing source files and other associated information.

C++

The C++ Standard does not specify this level of detail (although it does refer to end-of-line indicators, 2.1p1n1).

Common Implementations

Unicode Technical Report #13: "Unicode newline guidelines" discusses the issues associated with representing new-lines in files. The ISO 6429 standard also defines NEL (NExt Line, hexadecimal 0x85) as an end-of-line indicator. The Microsoft Windows convention is to indicate this end-of-line with a carriage return/line feed pair, \r\n (a convention that goes back through CP/M to DEC RT-11); the Unix convention is to use a single line feed character \n; the MacIntosh convention is to use the carriage return character, \r.

Some mainframes implement a form of text files that mimic punched cards by having fixed-length lines. Each line contains the same number of characters, often 80. The space after the last user-written character is sometimes padded with spaces, other times it is padded with null characters.

---

this International Standard treats such an end-of-line indicator as if it were a single new-line character. 225

Commentary

translation phase 1

The standard is not interested in the details of the byte representation of end-of-line on storage media. It makes use of the concept of end-of-line and uses the conceptual simplification of treating it as if it were a single character.

C++

2.1p1n1 *. . . (introducing new-line characters for end-of-line indicators) . . .*

---

basic execution character set control characters

In the basic execution character set, there shall be control characters representing alert, backspace, carriage return, and new line. 226

Commentary

This is a requirement on the implementation.

basic execution character set 221
translation phase 1
escape sequence syntax

These characters form part of the set of 96 execution character set members (counting the null character) defined by the standard, plus new line which is introduced in translation phase 1. However, these characters are not in the basic source character set, and are represented in it using escape sequences.

Other Languages

Few other languages include the concept of control characters, although many implementations provide semantics for them in source code (they are usually mapped exactly from the source to the execution character set). Java defines the same control characters as C and gives them their equivalent Ascii values. However, it does not define any semantics for these characters.

Common Implementations

ECMA-48 Control Functions for Coded Character Sets, Fifth Edition (available free from their Web site, http://www.ecma-international.ch) was fast-tracked as the third edition of ISO/IEC 6429. This standard defines significantly more control functions than those specified in the C Standard.

If any other characters are encountered in a source file (except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token), the behavior is undefined. 227

Commentary

The standard does not prohibit such characters from occurring in a source file outright. The Committee was aware of implementations that used such characters to extend the language. For instance, the use of the @ character in an object definition to specify its address in storage.

operator
preprocessing token converted to token

The list of exceptions is extensive. The only usage remaining, for such characters, is as a punctuator. *Any other character* has to be accepted as a preprocessing token. It may subsequently, for instance, be stringized. It is the attempt to convert this preprocessing token into a token where the undefined behavior occurs.

C90

Support for additional characters in identifiers is new in C99.

C++

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

The C++ Standard specifies the behavior and a translator is required to handle source code containing such a character. A C translator is permitted to issue a diagnostic and fail to translate the source code.

Other Languages

Most languages regard the appearance of an unknown character in the source as some form of error. Like C, most language implementations support additional characters in string literals and comments.

Common Implementations

Most implementations generate a diagnostic, either when the preprocessing token containing one of these characters is converted to a token, or as a result of the very likely subsequent syntax violation. Some implementations[3] define the @ character to be a token, its usual use being to provide the syntax for specifying the address at which an object is to be placed in storage. It is generally followed by an integer constant expression.

Coding Guidelines

An occurrence of a character outside of the basic source character set, in one of these contexts, is most likely to be a typing mistake and is very likely to be diagnosed by the translator. The other possibility is that such characters were intended to be used because use is being made of an extension. This issue is discussed elsewhere. ?? extensions cost/benefit

Example

```
1   static int glob @ 0x100; /* Put glob at location 0x100. */
```

---

228 A *letter* is an uppercase letter or a lowercase letter as defined above; letter

Commentary

This defines the term *letter*.

There is a third kind of case that characters can have, *titlecase* (a term sometimes applied to words where the first letter is in uppercase, or titlecase, and the other letters are in lowercase). In most instances titlecase is the same as uppercase, but there are a few characters where this is not true; for instance, the titlecase of the Unicode character U01C9, *lj*, is U01C8, *Lj*, and its uppercase is U01C7, *LJ*.

C90

This definition is new in C99.

---

229 in this International Standard the term does not include other characters that are letters in other alphabets.

Commentary

All implementations are required to support the basic source character set to which this terminology applies. Annex D lists those universal character names that can appear in identifiers. However, they are not referred to as letters (although they may well be regarded as such in their native language).

The term *letter* assumes that the orthography (writing system) of a language has an alphabet. Some orthographies, for instance Japanese, don't have an alphabet as such (let alone the concept of upper- and lowercase letters). Even when the orthography of a language does include characters that are considered to be matching upper and lowercase letters by speakers of that language (e.g., æ and Æ, å and Å), the C Standard does not define these characters to be *letter*s. orthography

C++

The definition used in the C++ Standard, 17.3.2.1.3 (the footnote applies to C90 only), implies this is also true in C++.

Coding Guidelines

The term *letter* has a common usage meaning in a number of different languages. Developers do not often use this term in its C Standard sense. Perhaps the safest approach for coding guideline documents to take is to avoid use of this term completely.

The universal character name construct provides a way to name other characters. 230

Commentary

ISO 10646 In theory all characters on planet Earth and beyond. In practice, those defined in ISO 10646.

C90

Support for universal character names is new in C99.

Other Languages

Other language standards are slowly moving to support ISO 10646. Java supports a similar concept.

Common Implementations

Support for these characters is relatively new. It will take time before similarities between implementations become apparent.

Forward references: universal character names (6.4.3), character constants (6.4.4.4), preprocessing directives (6.10), string literals (6.4.5), comments (6.4.9), string (7.1.1). 231

References

1. J. P. Beldie, S. Pastoor, and E. Schwarz. Fixed versus variable letter width for television text. *Human Factors*, 25(3):273–277, 1983.
2. A. Computer. *MrC/MrCpp C/C++ Compiler for Power Macintosh*. Apple Computer, Inc, 1.0 edition, 1995.
3. Keil. *C Compiler manual*. Keil Software, Inc, ??? edition, May 2005.
4. B. Kelk. Letter frequency rankings for various languages. www.bckelk.uklinux.net/words/etaoin.html, 2003.