The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.4.7 Header names

header name syntax

918

```
header-name:
                < h-char-sequence >
                " q-char-sequence "
h-char-sequence:
                h-char
                h-char-sequence h-char
h-char:
                any member of the source character set except
                                the new-line character and >
q-char-sequence:
                q-char
                q-char-sequence q-char
q-char:
                any member of the source character set except
                                the new-line character and "
```

Commentary

Headers enclosed between the < and > delimiters are commonly called a *system header* or *implementation header*. Such headers are generally special in that they are usually supplied by the implementation, host OS, or third-party API vendor. The " delimited form is commonly called a *header*.

Other Languages

While many languages do not specify any kind of header name tokens, their implementations usually add support for some such functionality as an extension. Use of double-quotes is commonly seen.

Usage

source file inclusion

Header name usage information is given elsewhere.

Semantics

The sequences in both forms of header names are mapped in an implementation-defined manner to headers or external source file names as specified in 6.10.2. 919

Commentary

header name 924 recognized within #include

source file inclusion

The *header-name* preprocessing token can only occur as part of a **#include** preprocessing directive. The issues involved in the implementation-defined mapping are discussed elsewhere.

---

characters between < and >delimiters

If the characters ', \, ", //, or /* occur in the sequence between the < and > delimiters, the behavior is undefined. 920

Commentary

The character sequences // and /* denote the start a comment. The character \ denotes the start of an escape sequence. The characters ' and " delimit character constants and string literals, respectively.

translation phase 3

translation phase 4

Preprocessing tokens and comments are handled in translation phase 3. The standard specifies no ordering dependencies on these operations. A lexical analyzer which has no knowledge of the context in which sequences of characters occur, would return a sequence of preprocessing tokens that subsequent processing (e.g., in translation phase 4) would need to join together to form a *header-name* preprocessing token. The characters listed above could all cause behavior such that a translator would not be able to join the necessary preprocessing tokens together in translation phase 4.

C90

The character sequence // was not specified as causing undefined behavior in C90 (which did not treat this sequence as the start of a comment).

Other Languages

Because support for some form of **#include** directive is usually provided as an extension by language implementations, the issue of certain sequences of characters having special meaning within a header name is part of an implementation's behavior, not the language specification.

Common Implementations

Most translators maintain sufficient context information that they are aware that a sequence of characters occurs on the same line as a **#include** preprocessing directive. In this case translators have sufficient information to know that a *header-name* preprocessing token is intended and can act accordingly.

Coding Guidelines

The mapping issues involved with these characters are discussed elsewhere.

translation phase 1
#include h-char-sequence

---

921 Similarly, if the characters ', \, //, or /* occur in the sequence between the " delimiters, the behavior is undefined.[69)]

Commentary

The issues involved are the same as those discussed in the previous sentence.

920 characters between < and >delimiters

C90

The character sequence // was not specified as causing undefined behavior in C90 (which did not treat this sequence as the start of a comment).

Coding Guidelines

One difference between the " delimiter and the < and > delimiters is that in the former case developers are likely to have some control over the characters that occur in the *q-char-sequence*.

920 characters between < and >delimiters

---

922 67) These tokens are sometimes called "digraphs".

footnote 67
digraphs

Commentary

There is no other term in common usage; in fact many developers are not aware of the existence of digraphs.

---

923 68) Thus [and <: behave differently when "stringized" (see 6.10.3.2), but can otherwise be freely interchanged.

footnote 68

Commentary

All six digraphs and their respective equivalent tokens behave differently when stringized. Token glueing is not an issue because there is only one situation where it might be possible to glue two digraphs together to form another meaningful token, and the Committee has already specified that this case does not create a valid token. The term *stringize* is not defined by the standard. However, its common usage is as the name of the # operator.

operator
EXAMPLE # ### #
operator

Coding Guidelines

Stringizing is a relatively rare operation and use of digraphs is even rarer. A guideline recommendation covering this case does not appear to be worthwhile.

Example

```
#define MK_STR(x) #x

char *p1 = MK_STR([);    /* Assigns the string "[" */
char *p2 = MK_STR(\??(); /* Assigns the string "[" */
char *p3 = MK_STR(<#);   /* Assigns the string "<#" */
```

header name recognized within #include

~~A header name preprocessing token is~~ Header name preprocessing tokens are recognized only within a **#include** preprocessing ~~directive.~~ directives or in implementation-defined locations within **#pragma** directives[DR324)]. 924

Commentary

header name exception to rule

The consequences of this requirement are discussed elsewhere.

The wording was changed by the response to DR #324.

C90

This statement summarizes the response to DR #017q39 against the C90 Standard.

~~C++~~

The ~~C++~~ Standard contains the same wording as the C90 Standard.

Common Implementations

All known translators implemented this requirement, even though it was not what C90 originally, technically specified. Most implementations simply take all characters on the line to the right of a **#include** and do their own special processing on it. In other contexts characters are processed through the usual preprocessing token creation machinery.

EXAMPLE tokenization

EXAMPLE The following sequence of characters: 925

```
0x3<1/a.h>1e2
#include <1/a.h>
#define const.member@$
```

forms the following sequence of preprocessing tokens (with each individual preprocessing token delimited by a *{* on the left and a *}* on the right).

```
{0x3}{<}{1}{/}{a}{.}{h}{>}{1e2}
{#}{include} {<1/a.h>}
{#}{define} {const}{.}{member}{@}{$}
```

Commentary

The tokenization formed for the character sequence `member@$` may be different when using implementations that support additional characters within an identifier preprocessing token.

Forward references: source file inclusion (6.10.2). 926

References