The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.10.3.2 The # operator

Constraints

operator

Each # preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list. 1950

Commentary

Within a replacement list the **#** preprocessing token is a unary operator. It is commonly called the *stringize* operator.

This operator is intended to be applied to the unexpanded form of the corresponding argument. Other preprocessing tokens in the replacement list can be *stringized* by simply delimiting them in double-quotes. Occurrences of a **#** preprocessing token that are not followed by a parameter are probable unintended and this constraint requirement ensures that translators will issue a diagnostic.

Usage

Based on the visible form of the `.c` files 0.26% (0.09% `.h` files) of the replacement lists of macro definitions contained a **#** operator. There were no obvious patterns to the usage.

Semantics

stringize operand

If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. 1951

Commentary

Rationale

Some pre-C89 implementations decided to replace identifiers found within a string literal if they matched a macro argument name. The replacement text is a "stringized" form of the actual argument token sequence. This practice appears to be contrary to K&R's definition of preprocessing in terms of token sequences. The C89 Committee declined to elaborate the syntax of string literals to the point where this practice could be condoned; however, since the facility provided by this mechanism seems to be widely used, the C89 Committee introduced a more tractable mechanism of comparable power.

In the following example:

```
1  #define HASH #
2
3  #define M(a) HASH a /* Expands to # a */
```

the **#** preprocessing token exists in the expanded replacement list, rather than the replacement list and is considered to be a punctuator rather than an operator.

Common Implementations

Microsoft C supports the preprocessor operator `#@` (call the *charizing* operator) as an extension. It converts its operand to a character constant.

Example

```
1  #define mkstr(a) # a
2
3  char *p_1 = mkstr("mnp\a");    /* Assigns "\"mnp\\a\"" */
4  char *p_2 = mkstr(000);        /* Assigns "000"        */
5  char *p_3 = mkstr(0);          /* Assigns "0"          */
6  char *p_4 = mkstr(0004);       /* Assigns "0004"       */
```

```
7   char *p_5 = mkstr(0.001E+000); /* Assigns "0.001E+000" */
8   char *p_6 = mkstr(x\
9   yz);                           /* No new-line in created string, i.e., "xyz" */
10  char *p_7 = mkstr(0
11  K);                            /* Assigns "0 K" */
```

1952 Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal.

white space between macro argument tokens

Commentary

Whether multiple white space between preprocessing tokens has already been converted to a single white space before this conversion is discussed elsewhere. Also there is no white space added where none existed in the source file.

white-space sequence replaced by one

Rationale

One problem with defining the effect of stringizing is the treatment of white space occurring in macro definitions. Where this could be discarded in the past, now upwards of one logical line may have to be retained. As a compromise between token-based and character-based preprocessing disciplines, the C89 Committee decided to permit white space to be retained as one bit of information: none or one. Arbitrary white space is replaced in the string by one space character.

Coding Guidelines

Any misconceptions about the white space will appear between preprocessing tokens that have been stringized is a developer education issue, not a coding guideline issue.

Example

The following assigns the string literal "`x y \\a+ 1`" to `p`.

```
1  #define mkstr(a) # a
2
3  char *p = mkstr(x y   \a+
4                          1);
```

Your author cannot think of a way to generate multiple, adjacent, space characters in a stringized sequence of preprocessing tokens.

1953 White space before the first preprocessing token and after the last preprocessing token composing the argument is deleted.

Commentary

This specification mirrors the one given for the replacement list and it is given for the same reasons.

white-space before/after replacement list

1954 Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants: a \ character is inserted before each " and \ character of a character constant or string literal (including the delimiting " characters), except that it is implementation-defined whether a \ character is inserted before the \ character beginning a universal character name.

escape sequence handling

Commentary

This specification is intended to ensure that the output produced by passing the string produced by the stringize operator as an argument to `printf`, for instance, is the same as the visible form (with white-space characters reduced to a single space character) of the preprocessing token sequence immediately prior to being stringized (although this sequence may not exist in the visible source). In the following example:

```
#define mkstr(s) #s

char *at = mkstr(\u0040);
```

translation phase 1
translation phase 5

if UCNs are mapped in translation phase 1 and @ is a supported character then the invocation of `mkstr` expands to "@". Otherwise the conversion occurs in translation phase 5 and the invocation expands to "\\u0040".

C90

Support for universal character names is new in C99.

C++

Support for universal character names is available in C++. However, wording for this clause of the C++ Standard was copied from C90, which did not support universal character names. The behavior of a C++ translator can be viewed as being equivalent to another C99 translator, in this regard. A C++ translator is not required to document its handling of a \ character before a universal character name.

Example

character ' or " matches

The # operator provides a mechanism for producing defined behavior from what appears to be a sequence of preprocessing tokens having no guaranteed interpretation within the C Standard.

```
#define mkstr(x) #x

char *p = mkstr(');  /* A single quote might, subject to undefined behavior, be given a meaning. */
```

If the replacement that results is not a valid character string literal, the behavior is undefined. 1955

Commentary

string literal syntax

For instance, syntactically, occurrences of the backslash character in string literals are limited to escape sequences. In the following example:

```
#define mkstr(x) # x

char *p = mkstr(a \ b); /* "a \ b" violates the syntax of string literals */
```

the result of the # operator need not be "a \ b".

Common Implementations

translation phase 5

Most implementations simply create a sequence of characters. However, processing in subsequent phases of translation (e.g., conversion of escape sequences) may also result in undefined behavior.

The character string literal corresponding to an empty argument is "". 1956

Commentary

This specification is more consistent than the stringize operator not returning any preprocessing token for this case.

C90

An occurrence of an empty argument in C90 caused undefined behavior.

C++

Like C90, the behavior in C++ is not explicitly defined (some implementations e.g., Microsoft C++, do not support empty arguments).

---

and ## evaluation order

The order of evaluation of # and ## operators is unspecified. 1957

Commentary

The C committee chose not to specify the relative precedence of these operators. In:

```
#define STR_GLUE(a, b) # a ## b

char *p = STR_GLUE(1, 2);
```

if {1} is glued to {2} and then stringised the resulting preprocessing token is defined. However, stringizing {1} and then attempting to glue it to {2} does not yield a defined preprocessing token (the behavior is undefined).

When both operators occur in a replacement list, performing token gluing first would appear to give the highest probability of having a defined result, when a stringize operator is also present. However, there is no requirement that implementations use this order. There is no need to specify an evaluation order for the # operator because it is unary (the evaluation order or the ## operator is discussed elsewhere).

evaluation order

Coding Guidelines

An order dependency on the evaluation of the # and ## operators only exists when either of them could be applied to the same preprocessing token in a replacement list. Unlike full expression evaluation it is not possible to use parentheses to group operands with preprocessor operators. These operators have to be adjacent to the preprocessing tokens they operate on. However, this combination of events rarely occurs (there are no occurrences in the .c files) and thus a guideline recommendation is not considered worthwhile.

full expression

Example

It is possible to specify an ordering by breaking the operations out into separate macro definitions.

```
#define MK_STR(a) #a
#define GLUE(a, b) a ## b

char *p_1 = MK_STR(GLUE(1, 2));

#define STR_GLUE(a, b) MK_STR(a ## b)

char *p_2 = STR_GLUE(1, 2);
```

References