The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.10.3.3 The ## operator

Constraints

##
operator

A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition. 1958

Commentary

The ## preprocessing token is a binary operator, requiring two operands. This operator is often called the *token gluing*, *glue*, or *token pasting* operator. Unlike the # operator, there is no requirement that the ## operator only be applied to preprocessing tokens that are parameters.

Common Implementations

gcc supports an additional use for the ## preprocessing operator. If a macro parameter accepting a variable number of arguments is passed no arguments and a ## appears before its use in the replacement list, the preceding comma punctuator is removed from the expanded replacement list. For instance, in:

```
#define eprintf(format, args...)         \
      fprintf (stderr, format , ## args)

/* The invocation */

eprintf("failure!\n");

/* is expanded to:
 *
 *    fprintf(stderr, "failure!\n");
 *
 * rather than to:
 *
 *    fprintf(stderr, "failure!\n" , );
 */
```

The Plan 9 C compiler intentionally lacks support for the ## operator.[1]

Example

Readers might like to work out which of the following two assignments to max relies on undefined behavior and which is strictly conforming.

file_1.c

```
#define __CONCAT__(_A, _B) _A ## _B
#define __CONCAT_U__(_A) _A ## u
#define ULONG32_C(__c)  __CONCAT__(__CONCAT_U__(__c), l)
#define ULONG32_MAX   ULONG32_C(4294967295)

unsigned long max = ULONG32_MAX;
```

file_2.c

```
#define __XXCONCAT__(_A, _B) _A ## _B
#define __CONCAT__(_A, _B) __XXCONCAT__(_A, _B)
#define __XCONCAT_U__(_A) _A ## u
#define __CONCAT_U__(_A) __XCONCAT_U__(_A)
#define ULONG32_C(__c)  __CONCAT__(__CONCAT_U__(__c), l)
#define ULONG32_MAX   ULONG32_C(4294967295)

unsigned long max = ULONG32_MAX;
```

Table 1958.1: Occurrence of the ## preprocessor operator (as a percentage of all occurrences of that operator). The form , ## identifier is a gcc extension (described in the Common implementations subclause). Based on the visible form of the .c and .h files.

Preprocessing Token Sequence	%
identifier ## identifier	70.2
, ## identifier	24.2
identifier ## identifier ## identifier	15.7
others	4.8
integer-constant ## identifier	1.8
integer-constant ## identifier ## *integer-constant*	1.0
identifier ## *integer-constant*	1.0

Semantics

1959 If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence;

Commentary

In this case the argument will not have been macro expanded before this replacement occurs (so if it consists of more than one preprocessing token only the first, and/or the last, are operated on).

parameter
argument macro expanded

Rationale

Another facility relied on in much current practice but not specified in K&R is "token pasting," or building a new token by macro argument substitution. One pre-C89 implementation replaced a comment within a macro expansion by no characters instead of the single space called for in K&R . The C89 Committee considered this practice unacceptable.

As with "stringizing," the facility was considered desirable, but not the extant implementation of this facility, so the C89 Committee invented another preprocessing operator. The ## operator within a macro expansion causes concatenation of the tokens on either side of it into a new composite token.

The specification of this pasting operator is based on these principles:

- Paste operations are explicit in the source.
- The ## operator is associative.
- A formal parameter as an operand for ## is not expanded before pasting. The actual parameter is substituted for the formal parameter; but the actual parameter is not replaced. Given, for example

```
1  #define a(n) aaa ## n
2  #define b    2
```

  the expansion of a(b) is aaab, not aaa2 or aaan.
- A normal operand for ## is not expanded before pasting.
- Pasting does not cross macro replacement boundaries.
- The token resulting from a paste operation is subject to further macro expansion.

These principles codify the essential features of prior art and are consistent with the specification of the stringizing operator.

Example

```
1  #define GLUE(a, b) a ## b
2
3  double d = GLUE(3.4e, -3); /* surprise! -3 is two preprocessing tokens */
```

argument no tokens replaced by placemarker token

however, if an argument consists of no preprocessing tokens, the parameter is replaced by a *placemarker* preprocessing token instead.[148)] 1960

Commentary

This defines the term *placemarker*. The standard uses placemarker preprocessing tokens to describe an effect, an implementation need not represent them internally. The need for a placemarker preprocessing token occurs because the **##** operator does not cross replacement boundaries.

C90

The explicitly using the concept of a placemarker preprocessing token is new in C99.

C++

The explicit concept of a placemarker preprocessing token is new in C99 and is not described in C++.

Coding Guidelines

In C90, an argument that consisted of no preprocessing tokens resulted in undefined behavior. The extent to which developers made use of such arguments and the behavior they expected from such usage is not known. Whether the definition of a behavior, in C99, and the introduction of the placemarker concept is something that developers need to be made aware of is not known.

For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. 1961

Commentary

EXAMPLE 1966 # ## # EXAMPLE reexamination EXAMPLE # and ##

The preprocessing token **##** is only recognized as an operator when it appears in the replacement list of the macro definition. When it appears anywhere else it is a punctuator.

Table 1961.1: Possible results of concatenating, using the **##** operator, pairs of preprocessing tokens (the one appearing in the left column followed by the one appearing in the top row) where the result might be defined (*undefined* denotes undefined behavior).

	identifier	*pp-number*	*punctuator*	*string-literal*	*character-constant*
identifier	identifier	identifier or undefined	undefined	string-literal or undefined	character-constant or undefined
pp-number	pp-number	pp-number	pp-number or undefined	undefined	undefined
punctuator	pp-number or undefined	pp-number or undefined	punctuator or undefined	undefined	undefined
everything else	undefined	undefined	undefined	undefined	undefined

placemarker preprocessor

Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. 1962

Commentary

EXAMPLE placemarker

This specification handles the special case of an argument containing no preprocessing tokens.

C90

The concept of placemarker preprocessing tokens is new in the C99 Standard. The behavior of concatenating an empty argument with preprocessing token was not explicitly defined in C90, it was undefined behavior.

C++

Like C90, the behavior of concatenating an empty argument with preprocessing token is not explicitly defined in C++, it is undefined behavior.

Example

N418.c

```
#define PI 3.1416
#define  F f
#define  D     /* Expands into no preprocessing tokens. */
#define LD L
#define  glue(a, b) a ## b
#define xglue(a, b) glue(a, b)

/*
 * The following:
 */
float        f = xglue(PI, F);
double       d = xglue(PI, D);
long double ld = xglue(PI, LD);
/*
 * should expand into:
 */
float        f = 3.1416f;
double       d = 3.1416;
long double ld = 3.1416L;
```

1963 If the result is not a valid preprocessing token, the behavior is undefined.

if result not valid

Commentary

While behavior in later phases is also likely to be undefined (e.g., when the preprocessing token is converted to a token) and a constraint violation, this specification applies during preprocessing and removes the need to define the behavior of any subsequent operations involving the result.

translation phase 7 preprocessing token shall have lexical form

Common Implementations

Many implementations allow invalid preprocessing tokens to be created and to subsequently be operands of the ## operator. This behavior enables the creation of preprocessing tokens that need to be built out of three separate preprocessing tokens, where the result of concatenating any two of them is not a valid preprocessing token.

Coding Guidelines

This situation is not sufficiently commonly for a guideline recommendation to be worthwhile.

Example

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

extern void f(int p, GLUE(. , . , .));

#include GLUE(< , header, >)
```

The resulting token is available for further macro replacement. 1964

Commentary

EXAMPLE 1966
#

However, it is not available for use as a preprocessing operator. The result of concatenating two preprocessing tokens is not treated any differently than other preprocessing tokens.

---

evaluation order

The order of evaluation of ## operators is unspecified. 1965

Commentary

If all intermediate results are defined for all orders of evaluation, the final result will always be the same. However, in some cases the result is not defined for some orders of evaluation.

Coding Guidelines

It is not possible to use parentheses to define an order of evaluation (the operators have to be adjacent to the preprocessing tokens that they operate on). The only solution is to break the replacement list up into separate macro definitions, each performing a single operation.

Example

```
#define GLUE_3(x, y, z) x ## y ## z

GLUE_3(>, >, =)    /* Behavior defined for all orders of evaluation.         */
GLUE_3(1, . , e10) /* Behavior only defined if left most ## performed first. */
/* There are no cases where the behavior is only defined if right most ## performed first. */
```

EXAMPLE
#

EXAMPLE In the following fragment: 1966

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)

char p[] = join(x, y); // equivalent to
                       // char p[] = "x ## y"
```

The expansion produces, at various stages:

```
join(x, y)

in_between(x hash_hash y)

in_between(x ## y)

mkstr(x ## y)

"x ## y"
```

In other words, expanding **hash_hash** produces a new token, consisting of two adjacent sharp signs, but this new token is not the ## operator.

Commentary

This example was created by the response to DR #017q22.

C++

This example is the response to a DR against C90. While there has been no such DR in C++, it is to be expected that WG21 would provide the same response.

1967 148) Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.

footnote 148

Commentary

The standard does not specify any formal syntax for function-like macro invocations, let alone placemarker preprocessing tokens.

function-like macro followed by (

C90

Support for the concept of placemarker preprocessing tokens is new in C99.

C++

Support for the concept of placemarker preprocessing tokens is new in C99 and they are not described in the C++ Standard.

References

1. R. Pike. How to use the Plan 9 C compiler. In *Plan 9 Programmer's Manual*. AT&T Bell Laboratories, 1995.