The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.10.3.4 Rescanning and further replacement

rescanning

After all parameters in the replacement list have been substituted and # and ## processing has taken place, all placemarker preprocessing tokens are removed. 1968

Commentary

The concept of placemarkers is only needed during the processing of the ## operator.

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 does not exist in C++.

rescanned along with subsequent tokens

Then, the resulting preprocessing token sequence is rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace. 1969

Commentary

Unlike argument substitution, and # and ## operator processing, the replacement list is not processed in isolation from the rest of the source code. For instance, in:

```
#define M1(a) (a+1)
#define M2(b) b

int ei_1 = M2(M1)(17);   /* becomes int ei_1 = (17+1);   */
int ei_2 = (M2(M1))(17); /* becomes int ei_2 = (M1)(17); */
```

after the invocation of M2 is expanded, the resulting sequence is M1. The rest of the source code is (17);. Rescanning including this sequence of preprocessing tokens, in the assignment to ei_1, results in an invocation of the macro M1.

Rationale

The rescanning rules incorporate an ambiguity. Given the definitions

```
#define  f(a)  a*g
#define  g     f
```

it is clear (or at least unambiguous) that the expansion of f(2)(9) is 2*f(9), the f in the result being introduced during the expansion of the original f, and so is not further expanded.

However, given the definitions

```
#define f(a)  a*g
#define g(a)  f(a)
```

the expansion will to be either 2*f(9) or 2*9*g: there are no clear grounds for making a decision whether the f(9) token string resulting from the initial expansion of f and the examination of the rest of the source file should be considered as nested within the expansion of f or not. The C89 Committee intentionally left this behavior ambiguous as it saw no useful purpose in specifying all the quirks of preprocessing for such questionably useful constructs.

Coding Guidelines

There are situations where it is intended that a replacement list include preprocessing tokens from the rest of the source file. For instance, if the name of a function needs to be unconditionally mapped to another name an object-like macro needs to be used. However, after expansion the mapped name might invoke a function-like macro:

```
#define isprint __PRINT_PROPERTY
#define __PRINT_PROPERTY(x) (((x < 0) || (x > 127)) ? 0 : __IS_PRINT[x])
int (__PRINT_PROPERTY)(int);

void f(void)
{
_Bool a_printable = isprint('a');
}
```

1970 If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced.

macro being replaced found during rescan

Commentary

Rationale

A problem faced by many pre-C89 preprocessors is how to use a macro name in its expansion without suffering "recursive death." The C89 Committee agreed simply to turn off the definition of a macro for the duration of the expansion of that macro.

Coding Guidelines

Whether or not a macro expansion that depends on this recursion-breaking rule requires significantly greater effort to comprehend than other macro expansions, involving similar numbers of preprocessing tokens but no recursion breaking, is not known. Neither is it known whether an alternative set of macros, that did not depend on recursion breaking, would require less effort. Without this information it is not possible to estimate the cost/benefit of any guideline recommendations and none are made here.

Example

```
extern int M_1,
           M_2;

#define M_1 M_2
#define M_2 M_1

void f(void)
{
M_1 = M_2; /* Macros do not alter the behavior, i.e., M_2 is still assigned to M_1 */
}
```

```
#define short static short

short si; /* Expands to static short si; */
```

Also see elsewhere for examples.

EXAMPLE reexamination

1971 Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced.

Commentary

Indirect recursion, via other macro definitions, is treated the same as direct recursion.

Coding Guidelines

The reasoning here is the same as for the case of direct recursion.

Example

```
static int M_0 = 0;

#define M_0(x)     M_ ## x
#define M_1(x) x + M_0(0)
#define M_2(x) x + M_1(1)
#define M_3(x) x + M_2(2)
#define M_4(x) x + M_3(3)
#define M_5(x) x + M_4(4)

int f_1(void)
{
return M_0(1)(2)(3)(4)(5); /* Expands to:
                                          2 + M_0(3)(4)(5)
                                                 or
                                          2 + M_0(0)(3)(4)(5) */
}

int f_2(void)
{
return M_0(5)(4)(3)(2)(1); /* Expands to: 4 + 4 + 3 + 2 + 1 + M_0(3)(2)(1) */
}
```

These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced. 1972

Commentary

The C preprocessor model is one where the incoming preprocessing tokens are processed and then output. It is not intended that the preprocessor have to hold on to all preprocessing tokens it has processed, until the end of the source file is reached.

```
#define F(a) a
#define FUNC(a) (a+1)

void f(void)
{
/*
 * The preprocessor works successively through the input without
 * backing up through previous processed preprocessing tokens.
 */
F(FUNC) FUNC (3); /* final token sequence is FUNC(3+1) */
}
```

This rule allows implementations to mark preprocessing tokens with a single bit (this bit is often referred to using the term *blue paint*, after the marking ink used by engineers, by members of the C committee), indicating that they are no longer available for replacement.

Example

```
#define A   A B C
#define B   B C A
```

```
#define C   C A B

A
/*
 * Using the notation:
 *    X={ }       the result of expanding X.
 *    lowercase   an identifier that has been 'painted blue'.
 * 'simplify'
   expand    A={ A B C }
   paint     A={ a B C }
   rescan    A={ a B={ B C A } C }
   paint     A={ a B={ b C a } C }
   rescan    A={ a B={ b C={ C A B } a } C }
   paint     A={ a B={ b C={ c a b } a } C }
             A={ a B={ b c a b a } C }
             A={ a b c a b a C }
   rescan    A={ a b c a b a C={ C A B }}
   paint     A={ a b c a b a C={ c a B }}
   rescan    A={ a b c a b a C={ c a B={ B C A }}}
   paint     A={ a b c a b a C={ c a B={ b c a }}}
             A={ a b c a b a C={ c a b c a }}
             A={ a b c a b a c a b c a }

   simplify  a b c a b a c a b c a

 * Final tokens output:

   A B C A B A C A B C A
 */
```

1973 The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 6.10.9 below.

expanded token sequence not treated as a directive

Commentary

As described elsewhere, a preprocessing directive is a particular sequence of preprocessing tokens at the start of translation phase 4. Macro-replaced preprocessing tokens don't exist until after the start of translation phase 4. The issue of arguments that resemble preprocessing directives is discussed elsewhere. The **_Pragma** unary operator is discussed elsewhere.

preprocessing directive consists of

argument resemble preprocessing directive

_Pragma operator

C90

Support for **_Pragma** unary operator expressions is new in C99.

C++

Support for **_Pragma** unary operator expressions is new in C99 and is not available in C++.

Example

```
#define H #
#define D define

#define DEFINE(a, b) H D a b

DEFINE(X, 3)
```

the invocation results in the sequence of preprocessing tokens:

{#} {define} {X} {3}

which are not treated as a preprocessing directive.

References