The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.10.3.1 Argument substitution

argument substitution

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. 1945

Commentary

The term *argument substitution* is also commonly used by developers to refer to this process.

How a sequence of preprocessing tokens within a source file are split into the arguments that belong to a particular function-like macro invocation is discussed elsewhere.

macro arguments separated by comma

Coding Guidelines

Experience suggests that many developers do not have accurate knowledge of the sequence of operations that occur during argument substitution. In many cases this lack of knowledge is not significant because the final result is as expected. In more subtle cases the results may be surprising. However, your author is not aware of any pattern to these developer misconceptions and so is unable to word any guideline recommendation. Given that there are few cases where a detailed knowledge of the sequence of operations is needed education may not help (i.e., without practice developers are unlikely to remember what they learned some time ago). Developer uncertainty about the sequence of events may lead them to select the most likely behavior from among a range of possibilities they consider to be plausible in the given context. There are no obvious guideline recommendations covering this pattern of usage.

macro arguments separated by comma

parameter argument macro expanded

A parameter in the replacement list, unless preceded by a # or ## preprocessing token or followed by a ## preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. 1946

Commentary

That is, parameters occurring in the replacement list are replaced after their corresponding arguments have been macro expanded. The # and ## operator contexts are the only situations where any macros appearing in a parameter's corresponding argument are not considered for replacement. In the following the expanded form of PARAM is not examined for preprocessing tokens that have the same spelling as a parameter of F.

```
#define PARAM param
#define F(param) param + PARAM

int glob = F(1); /* Expands to 1+param, rather than 1+1 */
```

Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the preprocessing file; 1947

Commentary

Each argument is expanded in isolation (i.e., there is no interaction between arguments or any other preprocessing tokens in the source file).

Example

In the following the argument in the invocation of FM2 is expanded to the two tokens {FM1} and {(}.

```
#define L_PAREN (
#define FM1(a) a
#define FM2(b) b 23)

void f1(void)
{
FM2(FM1 L_PAREN);
}
```

Completely expanding the argument requires that FM1 then be expanded (it is followed by an opening parentheses). However, this expansion does not succeed because there is no matching closing parentheses, in the sequence of preprocessing tokens for that argument. The behavior is undefined. Continuing on from the above source:

```
1  void f2(void)
2  {
3  FM2(FM1(L_PAREN));
4  }
```

FM2 expands to (23). The following definition achieves what may have been intended:

```
1  #define FM3(c, d) c d 23)
2
3  void f3(void)
4  {
5  FM3(FM1, L_PAREN);
6  }
```

the invocation of FM3 expands to 23 (the argument FM1 is not expanded further, as an argument, because it is not followed by an opening parentheses).

1948 no other preprocessing tokens are available.

Commentary

In particular any subsequent preprocessing tokens from the source file, or any preprocessing tokens following the parameter in the replacement list.

1949 An identifier **__VA_ARGS__** that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it.

Commentary

This is a requirement on the implementation.

This is replaced by all the arguments that match the ellipsis, including the commas between them. Rationale

The extent to which the arguments corresponding to the parameter **__VA_ARGS__** are replaced may be affected by the presence of commas. For instance, in:

```
1  #define LPAREN (
2  #define RPAREN )
3  #define F(x, y) x + y
4  #define ELLIP_FUNC(...) __VA_ARGS__
5
6  ELLIP_FUNC(F, LPAREN, 'a', 'b', RPAREN); /* 1st invocation */
7  ELLIP_FUNC(F  LPAREN  'a', 'b'  RPAREN); /* 2nd invocation */
```

the argument in the second invocation of ELLIP_FUNC expands to F('a', 'b') which in turn expands to 'a'+'b'. This expansion sequence does not occur in the first invocation because of the comma separating F from ((and other preprocessing tokens).

C90

Support for **__VA_ARGS__** is new in C99.

C++

Support for **__VA_ARGS__** is new in C99 and is not specified in the C++ Standard.

References