3.6 Variadic Macros

A macro can be declared to accept a variable number of arguments much as a function can. The syntax for defining the macro is similar to that of a function. Here is an example:

```
#define eprintf(...) fprintf (stderr, __VA_ARGS__)
```

This kind of macro is called variadic. When the macro is invoked, all the tokens in its argument list after the last named argument (this macro has none), including any commas, become the variable argument. This sequence of tokens replaces the identifier `__VA_ARGS__` in the macro body wherever it appears. Thus, we have this expansion:

```
eprintf ("%s:%d: ", input_file, lineno)
     →  fprintf (stderr, "%s:%d: ", input_file, lineno)
```

The variable argument is completely macro-expanded before it is inserted into the macro expansion, just like an ordinary argument. You may use the ‘`#`’ and ‘`##`’ operators to stringize the variable argument or to paste its leading or trailing token with another token. (But see below for an important special case for ‘`##`’.)

If your macro is complicated, you may want a more descriptive name for the variable argument than `__VA_ARGS__`. CPP permits this, as an extension. You may write an argument name immediately before the ‘`...`’; that name is used for the variable argument. The `eprintf` macro above could be written

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

using this extension. You cannot use `__VA_ARGS__` and this extension in the same macro.

You can have named arguments as well as variable arguments in a variadic macro. We could define `eprintf` like this, instead:

```
#define eprintf(format, ...) fprintf (stderr, format, __VA_ARGS__)
```

This formulation looks more descriptive, but historically it was less flexible: you had to supply at least one argument after the format string. In standard C, you could not omit the comma separating the named argument from the variable arguments. (Note that this restriction has been lifted in C++2a, and never existed in GNU C; see below.)

Furthermore, if you left the variable argument empty, you would have gotten a syntax error, because there would have been an extra comma after the format string.

```
eprintf("success!\n", );
     → fprintf(stderr, "success!\n", );
```

This has been fixed in C++2a, and GNU CPP also has a pair of extensions which deal with this problem.

First, in GNU CPP, and in C++ beginning in C++2a, you are allowed to leave the variable argument out entirely:

```
eprintf ("success!\n")
     → fprintf(stderr, "success!\n", );
```

Second, C++2a introduces the `__VA_OPT__` function macro. This macro may only appear in the definition of a variadic macro. If the variable argument has any tokens, then a `__VA_OPT__` invocation expands to its argument; but if the variable argument does not have any tokens, the `__VA_OPT__` expands to nothing:

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

`__VA_OPT__` is also available in GNU C and GNU C++.

Historically, GNU CPP has also had another extension to handle the trailing comma: the ‘`##`’ token paste operator has a special meaning when placed between a comma and a variable argument. Despite the introduction of `__VA_OPT__`, this extension remains supported in GNU CPP, for backward compatibility. If you write

```
#define eprintf(format, ...) fprintf (stderr, format, ##__VA_ARGS__)
```

and the variable argument is left out when the `eprintf` macro is used, then the comma before the ‘`##`’ will be deleted. This does not happen if you pass an empty argument, nor does it happen if the token preceding ‘`##`’ is anything other than a comma.

```
eprintf ("success!\n")
     → fprintf(stderr, "success!\n");
```

The above explanation is ambiguous about the case where the only macro parameter is a variable arguments parameter, as it is meaningless to try to distinguish whether no argument at all is an empty argument or a missing argument. CPP retains the comma when conforming to a specific C standard. Otherwise the comma is dropped as an extension to the standard.

The C standard mandates that the only place the identifier `__VA_ARGS__` can appear is in the replacement list of a variadic macro. It may not be used as a macro name, macro argument name, or within a different type of macro. It may also be forbidden in open text; the standard is ambiguous. We recommend you avoid using it except for its defined purpose.

Likewise, C++ forbids `__VA_OPT__` anywhere outside the replacement list of a variadic macro.

Variadic macros became a standard part of the C language with C99. GNU CPP previously supported them with a named variable argument (‘`args...`’ , not ‘`...`’ and `__VA_ARGS__`), which is still supported for backward compatibility.

Next: Predefined Macros, Previous: Concatenation, Up: Macros [Contents][Index]