piled in "debugging mode," with extra statements included to produce debugging output:

```
#define DEBUG
```

Incidentally, it's legal for a macro's replacement list to be empty, as this example shows.

When macros are used as constants, C programmers customarily capitalize all letters in their names. However, there's no consensus as to how to capitalize macros used for other purposes. Since macros (especially parameterized macros) can be a source of bugs, some programmers like to draw attention to them by using all upper-case letters in their names. Others prefer lower-case names, following the style of Kernighan and Ritchie's *The C Programming Language*.

Parameterized Macros

The definition of a ***parameterized macro*** (also known as a ***function-like macro***) has the form

#define directive (parameterized macro)

`#define` *identifier* (x_1 , x_2 , ... , x_n) *replacement-list*

where x_1, x_2, ..., x_n are identifiers (the macro's ***parameters***). The parameters may appear as many times as desired in the replacement list.

There must be *no space* between the macro name and the left parenthesis. If space is left, the preprocessor will assume that we're defining a simple macro; it will treat (x_1 , x_2 , ... , x_n) as part of the replacement list.

When the preprocessor encounters the definition of a parameterized macro, it stores the definition away for later use. Wherever a macro ***invocation*** of the form *identifier* (y_1 , y_2 , ... , y_n) appears later in the program (where y_1, y_2, ..., y_n are sequences of tokens), the preprocessor replaces it with *replacement-list*, substituting y_1 for x_1, y_2 for x_2, and so forth.

For example, suppose that we've defined the following macros:

```
#define MAX(x,y)    ((x)>(y)?(x):(y))
#define IS_EVEN(n)  ((n)%2==0)
```

(The number of parentheses in these macros may seem excessive, but there's a reason, as we'll see later in this section.) Now suppose that we invoke the two macros in the following way:

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```