The preprocessor will replace these lines by

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

As this example shows, parameterized macros often serve as simple functions. `MAX` behaves like a function that computes the larger of two values. `IS_EVEN` behaves like a function that returns 1 if its argument is an even number and 0 otherwise.

Here's a more complicated macro that behaves like a function:

```
#define TOUPPER(c) ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
```

This macro tests whether the character `c` is between `'a'` and `'z'`. If so, it produces the upper-case version of `c` by subtracting `'a'` and adding `'A'`. If not, it leaves `c` unchanged. (The `<ctype.h>` header provides a similar function named `toupper` that's more portable.)

<ctype.h> header ➤ 23.5

A parameterized macro may have an empty parameter list. Here's an example:

```
#define getchar() getc(stdin)
```

The empty parameter list isn't really needed, but it makes `getchar` resemble a function. (Yes, this is the same `getchar` that belongs to `<stdio.h>`. We'll see in Section 22.4 that `getchar` is usually implemented as a macro as well as a function.)

Using a parameterized macro instead of a true function has a couple of advantages:

- ***The program may be slightly faster.*** A function call usually requires some overhead during program execution—context information must be saved, arguments copied, and so forth. A macro invocation, on the other hand, requires no run-time overhead. (Note, however, that C99's inline functions provide a way to avoid this overhead without the use of macros.)

C99
Inline functions ➤ 18.6

- ***Macros are "generic."*** Macro parameters, unlike function parameters, have no particular type. As a result, a macro can accept arguments of any type, provided that the resulting program—after preprocessing—is valid. For example, we could use the `MAX` macro to find the larger of two values of type `int`, `long`, `float`, `double`, and so forth.

But parameterized macros also have disadvantages:

- ***The compiled code will often be larger.*** Each macro invocation causes the insertion of the macro's replacement list, thereby increasing the size of the source program (and hence the compiled code). The more often the macro is used, the more pronounced this effect is. The problem is compounded when macro invocations are nested. Consider what happens when we use `MAX` to find the largest of three numbers:

```
n = MAX(i, MAX(j, k));
```

Here's the same statement after preprocessing:

```
n = ((i)>(((j)>(k)?(j):(k)))?(i):(((j)>(k)?(j):(k))));
```