- *Arguments aren't type-checked.* When a function is called, the compiler checks each argument to see if it has the appropriate type. If not, either the argument is converted to the proper type or the compiler produces an error message. Macro arguments aren't checked by the preprocessor, nor are they converted.
- *It's not possible to have a pointer to a macro.* As we'll see in Section 17.7, C allows pointers to functions, a concept that's quite useful in certain programming situations. Macros are removed during preprocessing, so there's no corresponding notion of "pointer to a macro"; as a result, macros can't be used in these situations.
- *A macro may evaluate its arguments more than once.* A function evaluates its arguments only once; a macro may evaluate its arguments two or more times. Evaluating an argument more than once can cause unexpected behavior if the argument has side effects. Consider what happens if one of MAX's arguments has a side effect:

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

Here's the same line after preprocessing:

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

If `i` is larger than `j`, then `i` will be (incorrectly) incremented twice and `n` will be assigned an unexpected value.

Errors caused by evaluating a macro argument more than once can be difficult to find, because a macro invocation looks the same as a function call. To make matters worse, a macro may work properly most of the time, failing only for certain arguments that have side effects. For self-protection, it's a good idea to avoid side effects in arguments.

Parameterized macros are good for more than just simulating functions. In particular, they're often used as patterns for segments of code that we find ourselves repeating. Suppose that we grow tired of writing

```
printf("%d\n", i);
```

every time we need to print an integer `i`. We might define the following macro, which makes it easier to display integers:

```
#define PRINT_INT(n) printf("%d\n", n)
```

Once `PRINT_INT` has been defined, the preprocessor will turn the line

```
PRINT_INT(i/j);
```

into

```
printf("%d\n", i/j);
```