the time of substitution. As a result, `CONCAT(a,CONCAT(b,c))` expands to `aCONCAT(b,c)`, which can't be expanded further, since there's no macro named `aCONCAT`.

There's a way to solve the problem, but it's not pretty. The trick is to define a second macro that simply calls the first one:

```
#define CONCAT2(x,y) CONCAT(x,y)
```

Writing `CONCAT2(a,CONCAT2(b,c))` now yields the desired result. As the preprocessor expands the outer call of `CONCAT2`, it will expand `CONCAT2(b,c)` as well; the difference is that `CONCAT2`'s replacement list doesn't contain `##`. If none of this makes any sense, don't worry; it's not a problem that arises often.

The `#` operator has a similar difficulty, by the way. If `#x` appears in a replacement list, where `x` is a macro parameter, the corresponding argument is not expanded. Thus, if `N` is a macro representing `10`, and `STR(x)` has the replacement list `#x`, expanding `STR(N)` yields `"N"`, not `"10"`. The solution is similar to the one we used with `CONCAT`: defining a second macro whose job is to call `STR`.

*Q: **Suppose that the preprocessor encounters the original macro name during rescanning, as in the following example:**

```
#define N (2*M)
#define M (N+1)
i = N;   /* infinite loop? */
```

The preprocessor will replace `N` by `(2*M)`, then replace `M` by `(N+1)`. Will the preprocessor replace `N` again, thus going into an infinite loop? [p. 326]

A: Some old preprocessors will indeed go into an infinite loop, but newer ones shouldn't. According to the C standard, if the original macro name reappears during the expansion of a macro, the name is not replaced again. Here's how the assignment to `i` will look after preprocessing:

```
i = (2*(N+1));
```

Some enterprising programmers take advantage of this behavior by writing macros whose names match reserved words or functions in the standard library. Consider the `sqrt` library function. `sqrt` computes the square root of its argument, returning an implementation-defined value if the argument is negative. Perhaps we would prefer that `sqrt` return 0 if its argument is negative. Since `sqrt` is part of the standard library, we can't easily change it. We can, however, define a `sqrt` *macro* that evaluates to 0 when given a negative argument:

`sqrt` function ➤23.3

```
#undef sqrt
#define sqrt(x) ((x)>=0?sqrt(x):0)
```

A later call of `sqrt` will be intercepted by the preprocessor, which expands it into the conditional expression shown here. The call of `sqrt` inside the conditional expression won't be replaced during rescanning, so it will remain for the compiler