contains invocations of other macros (PI, in this case). The preprocessor will rescan the replacement list as many times as necessary to eliminate all macro names. Q&A

- ***The preprocessor replaces only entire tokens, not portions of tokens.*** As a result, the preprocessor ignores macro names that are embedded in identifiers, character constants, and string literals. For example, suppose that a program contains the following lines:

```
#define SIZE 256

int BUFFER_SIZE;

if (BUFFER_SIZE > SIZE)
  puts("Error: SIZE exceeded");
```

  After preprocessing, these lines will have the following appearance:

```
int BUFFER_SIZE;

if (BUFFER_SIZE > 256)
  puts("Error: SIZE exceeded");
```

  The identifier BUFFER_SIZE and the string "Error: SIZE exceeded" weren't affected by preprocessing, even though both contain the word SIZE.

- ***A macro definition normally remains in effect until the end of the file in which it appears.*** Since macros are handled by the preprocessor, they don't obey normal scope rules. A macro defined inside the body of a function isn't local to that function; it remains defined until the end of the file.
- ***A macro may not be defined twice unless the new definition is identical to the old one.*** Differences in spacing are allowed, but the tokens in the macro's replacement list (and the parameters, if any) must be the same.
- ***Macros may be "undefined" by the* #undef *directive.*** The #undef directive has the form

#undef directive

#undef *identifier*

where *identifier* is a macro name. For example, the directive

```
#undef N
```

removes the current definition of the macro N. (If N hasn't been defined as a macro, the #undef directive has no effect.) One use of #undef is to remove the existing definition of a macro so that it can be given a new definition.

Parentheses in Macro Definitions

The replacement lists in our macro definitions have been full of parentheses. Is it really necessary to have so many? The answer is an emphatic yes; if we use fewer