A shortage of parentheses in a macro definition can cause some of C's most frustrating errors. The program will usually compile and the macro will appear to work, failing only at the least convenient times.

Creating Longer Macros

The comma operator can be useful for creating more sophisticated macros by allowing us to make the replacement list a series of expressions. For example, the following macro will read a string and then print it:

```
#define ECHO(s) (gets(s), puts(s))
```

Calls of `gets` and `puts` are expressions, so it's perfectly legal to combine them using the comma operator. We can invoke ECHO as though it were a function:

```
ECHO(str);    /* becomes (gets(str), puts(str)); */
```

Instead of using the comma operator in the definition of ECHO, we could have enclosed the calls of `gets` and `puts` in braces to form a compound statement:

```
#define ECHO(s) { gets(s); puts(s); }
```

Unfortunately, this method doesn't work as well. Suppose that we use ECHO in an `if` statement:

```
if (echo_flag)
  ECHO(str);
else
  gets(str);
```

Replacing ECHO gives the following result:

```
if (echo_flag)
  { gets(str); puts(str); };
else
  gets(str);
```

The compiler treats the first two lines as a complete `if` statement:

```
if (echo_flag)
  { gets(str); puts(str); }
```

It treats the semicolon that follows as a null statement and produces an error message for the `else` clause, since it doesn't belong to any `if`. We could solve the problem by remembering not to put a semicolon after each invocation of ECHO, but then the program would look odd.

The comma operator solves this problem for ECHO, but not for all macros. Suppose that a macro needs to contain a series of *statements*, not just a series of *expressions*. The comma operator is of no help; it can glue together expressions.