Q & A

Q: I've seen programs that contain a # on a line by itself. Is this legal?

A: Yes. This is the *null directive;* it has no effect. Some programmers use null directives for spacing within conditional compilation blocks:

```
#if INT_MAX < 100000
#
#error int type is too small
#
#endif
```

Blank lines would also work, of course, but the # helps the reader see the extent of the block.

Q: I'm not sure which constants in a program need to be defined as macros. Are there any guidelines to follow? [p. 319]

A: One rule of thumb says that every numeric constant, other than 0 or 1, should be a macro. Character and string constants are problematic, since replacing a character or string constant by a macro doesn't always improve readability. I recommend using a macro instead of a character constant or string literal provided that (1) the constant is used more than once and (2) the possibility exists that the constant might someday be modified. Because of rule (2), I don't use macros such as

```
#define NUL '\0'
```

although some programmers do.

Q: What does the # operator do if the argument that it's supposed to "stringize" contains a " or \ character? [p. 324]

A: It converts " to \" and \ to \\. Consider the following macro:

```
#define STRINGIZE(x) #x
```

The preprocessor will replace `STRINGIZE("foo")` by `"\"foo\""`.

***Q: I can't get the following macro to work properly:**

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

`CONCAT(a,b)` gives `ab`, as expected, but `CONCAT(a,CONCAT(b,c))` gives an odd result. What's going on?

A: Thanks to rules that Kernighan and Ritchie call "bizarre," macros whose replacement lists depend on ## usually can't be called in a nested fashion. The problem is that `CONCAT(a,CONCAT(b,c))` isn't expanded in a "normal" fashion, with `CONCAT(b,c)` yielding `bc`, then `CONCAT(a,bc)` giving `abc`. Macro parameters that are preceded or followed by ## in a replacement list aren't expanded at