A: Yes, as the following example shows:

```
char fmt[] = "%d\n";
int i;
...
printf(fmt, i);
```

This ability opens the door to some intriguing possibilities—reading a format string as input, for example.

Q: If I want `printf` to write a string `str`, can't I just supply `str` as the format string, as in the following example?

```
printf(str);
```

A: Yes, but it's risky. If `str` contains the `%` character, you won't get the desired result, since `printf` will assume it's the beginning of a conversion specification.

***Q: How can `read_line` detect whether `getchar` has failed to read a character? [p. 287]**

A: If it can't read a character, either because of an error or because of end-of-file, `getchar` returns the value EOF, which has type `int`. Here's a revised version of `read_line` that tests whether the return value of `getchar` is EOF. Changes are marked in **bold:**

EOF macro ➤22.4

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n' && ch != EOF)
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';
  return i;
}
```

Q: Why does `strcmp` return a number that's less than, equal to, or greater than zero? Also, does the exact return value have any significance? [p. 292]

A: `strcmp`'s return value probably stems from the way the function is traditionally written. Consider the version in Kernighan and Ritchie's *The C Programming Language:*

```
int strcmp(char *s, char *t)
{
  int i;

  for (i = 0; s[i] == t[i]; i++)
    if (s[i] == '\0')
      return 0;
  return s[i] - t[i];
}
```