scanf always reads from stdin (the standard input stream), whereas fscanf reads from the stream indicated by its first argument:

```
scanf("%d%d", &i, &j);          /* reads from stdin */
fscanf(fp, "%d%d", &i, &j);     /* reads from fp */
```

A call of scanf is equivalent to a call of fscanf with stdin as the first argument.

The ...scanf functions return prematurely if an *input failure* occurs (no more input characters could be read) or if a *matching failure* occurs (the input characters didn't match the format string). (In C99, an input failure can also occur because of an *encoding error*, which means that an attempt was made to read a multibyte character, but the input characters didn't correspond to any valid multibyte character.) Both functions return the number of data items that were read and assigned to objects; they return EOF if an input failure occurs before any data items can be read.

C99

multibyte characters ➤25.2

Loops that test scanf's return value are common in C programs. The following loop, for example, reads a series of integers one by one, stopping at the first sign of trouble:

idiom

```
while (scanf("%d", &i) == 1) {
  ...
}
```

...scanf Format Strings

Calls of the ...scanf functions resemble those of the ...printf functions. That similarity can be misleading, however; the ...scanf functions work quite differently from the ...printf functions. It pays to think of scanf and fscanf as "pattern-matching" functions. The format string represents a pattern that a ...scanf function attempts to match as it reads input. If the input doesn't match the format string, the function returns as soon as it detects the mismatch; the input character that didn't match is "pushed back" to be read in the future.

A ...scanf format string may contain three things:

- *Conversion specifications.* Conversion specifications in a ...scanf format string resemble those in a ...printf format string. Most conversion specifications skip white-space characters at the beginning of an input item (the exceptions are %[, %c, and %n). Conversion specifications never skip *trailing* white-space characters, however. If the input contains •123¤, the %d conversion specification consumes •, 1, 2, and 3, but leaves ¤ unread. (I'm using • to represent the space character and ¤ to represent the new-line character.)

white-space characters ➤3.2

- *White-space characters.* One or more consecutive white-space characters in a ...scanf format string match zero or more white-space characters in the input stream.
- *Non-white-space characters.* A non-white-space character other than % matches the same character in the input stream.