join them into a single string. This rule allows us to split a string literal over two or more lines:

```
printf("When you come to a fork in the road, take it.  "
       "--Yogi Berra");
```

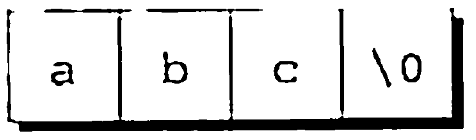
How String Literals Are Stored

We've used string literals often in calls of `printf` and `scanf`. But when we call `printf` and supply a string literal as an argument, what are we actually passing? To answer this question, we need to know how string literals are stored.

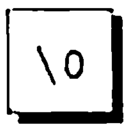
In essence, C treats string literals as character arrays. When a C compiler encounters a string literal of length *n* in a program, it sets aside *n* + 1 bytes of memory for the string. This area of memory will contain the characters in the string, plus one extra character—the ***null character***—to mark the end of the string. The null character is a byte whose bits are all zero, so it's represented by the `\0` escape sequence.

Don't confuse the null character (`'\0'`) with the zero character (`'0'`). The null character has the code 0; the zero character has a different code (48 in ASCII).

For example, the string literal `"abc"` is stored as an array of four characters (a, b, c, and `\0`):



String literals may be empty; the string `""` is stored as a single null character:



Since a string literal is stored as an array, the compiler treats it as a pointer of type `char *`. Both `printf` and `scanf`, for example, expect a value of type `char *` as their first argument. Consider the following example:

```
printf("abc");
```

When `printf` is called, it's passed the address of `"abc"` (a pointer to where the letter a is stored in memory).

Operations on String Literals

In general, we can use a string literal wherever C allows a `char *` pointer. For example, a string literal can appear on the right side of an assignment: