
STRINGS

String Literals

A ***string literal*** is a sequence of characters enclosed within double quotes:

```
"When you come to a fork in the road, take it."
```

We first encountered string literals in Chapter 2; they often appear as format strings in calls of `printf` and `scanf`.

Counting a string literal

If we find that a string literal is too long to fit conveniently on a single line, C allows us to continue it on the next line, provided that we end the first line with a backslash character (\). No other characters may follow \ on the same line, other than the (invisible) new-line character at the end:

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

In general, the \ character can be used to join two or more lines of a program into a single line (a process that the C standard refers to as "splicing"). We'll see more examples of splicing in Section 14.3.

The \ technique has one drawback: the string must continue at the beginning of the next line, thereby wrecking the program's indented structure. There's a bet-

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 are string literals stored?

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:

Operations on string literals

```
char *p;

p = "abc";
```

This assignment doesn't copy the characters in `"abc"`; it merely makes `p` point to the first character of the string.

C allows pointers to be subscripted, so we can subscript string literals:

```
char ch;

ch = "abc"[1];
```

The new value of `ch` will be the letter `b`. The other possible subscripts are 0 (which would select the letter `a`), 2 (the letter `c`), and 3 (the null character). This property of string literals isn't used that much, but occasionally it's handy. Consider the following function, which converts a number between 0 and 15 into a character that represents the equivalent hex digit:

```
char digit_to_hex_char(int digit)
{
  return "0123456789ABCDEF"[digit];
}
```

Attempting to modify a string literal causes undefined behavior:

```
char *p = "abc";

*p = 'd';    /*** WRONG ***/
```

A program that tries to change a string literal may crash or behave erratically.

String Variables & Initialization

idiom

```
#define STR_LEN 80
...
char str[STR_LEN+1];
```

A string variable can be initialized at the same time it's declared:

```
char date1[8] = "June 14";
```

String Initialization

The compiler will put the characters from "June 14" in the date1 array, then add a null character so that date1 can be used as a string. Here's what date1 will look like:

Although "June 14" appears to be a string literal, it's not. Instead, C views it as an abbreviation for an array initializer. In fact, we could have written

```
char date1[8] = {'J', 'u', 'n', 'e', ' ', '1', '4', '\0'};
```

I think you'll agree that the original version is easier to read.

What if the initializer is too short to fill the string variable? In that case, the compiler adds extra null characters. Thus, after the declaration

```
char date2[9] = "June 14";
```

date2 will have the following appearance:

String Initialization

```
char date3[7] = "June 14";
```

There's no room for the null character, so the compiler makes no attempt to store one:

The declaration of a string variable may omit its length, in which case the compiler computes it:

```
char date4[] = "June 14";
```

Character arrays vs Character pointers

Let's compare the declaration

```
char date[] = "June 14";
```

which declares `date` to be an *array*, with the similar-looking

```
char *date = "June 14";
```

```
char str[STR_LEN+1], *p;

p = str;
```

```
char *p;

p[0] = 'a';     /*** WRONG ***/
p[1] = 'b';     /*** WRONG ***/
p[2] = 'c';     /*** WRONG ***/
p[3] = '\0';    /*** WRONG ***/
```

Writing Strings: Printf & Puts

```
char str[] = "Are we having fun yet?";
printf("%s\n", str);
```

The output will be

```
Are we having fun yet?
```

```
printf("%.6s\n", str);
```

will print

```
Are we
```

`printf` isn't the only function that can write strings. The C library also provides `puts`, which is used in the following way:

```
puts(str);
```

Reading Strings: scanf & gets

The %s conversion specification allows scanf to read a string into a character array:

```
scanf("%s", str);
```

the user enters the line

```
  To C, or not to C: that is the question.
```

scanf will store the string "To" in sentence. The next call of scanf will resume reading the line at the space after the word To.

Now suppose that we replace scanf by gets:

```
gets(sentence);
```

When the user enters the same input as before, gets will store the string

```
"  To C, or not to C: that is the question."
```

Reading Sting Character by Character

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

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';        /* terminates string */
  return i;             /* number of characters stored */
}
```

Accessing the characters in array

```
int count_spaces(const char s[])
{
  int count = 0, i;

  for (i = 0; s[i] != '\0'; i++)
    if (s[i] == ' ')
      count++;
  return count;
}
```

Using the C string library

Direct attempts to copy or compare strings will fail. For example, suppose that `str1` and `str2` have been declared as follows:

```
char str1[10], str2[10];
```

Copying a string into a character array using the = operator is not possible:

```
str1 = "abc";  /*** WRONG ***/
str2 = str1;   /*** WRONG ***/
```

We saw in Section 12.3 that using an array name as the left operand of = is illegal. *Initializing* a character array using = is legal, though:

```
char str1[10] = "abc";
```

In the context of a declaration, = is not the assignment operator.

Attempting to compare strings using a relational or equality operator is legal but won't produce the desired result:

```
if (str1 == str2) …   /*** WRONG ***/
```

This statement compares `str1` and `str2` as *pointers*; it doesn't compare the contents of the two arrays. Since `str1` and `str2` have different addresses, the expression `str1 == str2` must have the value 0.

Using the C Sting library

strcpy ()

strncpy ()

strlen ()

strcat ()

strncat()

strcmp()

Arrays of Strings

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

Command line arguments

```
int main(int argc, char *argv[])
{
  …
}
```