Chapter 8: Strings

Strings in C are represented by arrays of characters. The end of the string is marked with a special character, the *null character*, which is simply the character with the value 0. (The null character has no relation except in name to the *null pointer*. In the ASCII character set, the null character is named NUL.) The null or string-terminating character is represented by another character escape sequence, `\0`. (We've seen it once already, in the `getline` function of chapter 6.)

Because C has no built-in facilities for manipulating entire arrays (copying them, comparing them, etc.), it also has very few built-in facilities for manipulating strings.

In fact, C's only truly built-in string-handling is that it allows us to use *string constants* (also called *string literals*) in our code. Whenever we write a string, enclosed in double quotes, C automatically creates an array of characters for us, containing that string, terminated by the `\0` character. For example, we can declare and define an array of characters, and initialize it with a string constant:

```
char string[] = "Hello, world!";
```

In this case, we can leave out the dimension of the array, since the compiler can compute it for us based on the size of the initializer (14, including the terminating `\0`). This is the only case where the compiler sizes a string array for us, however; in other cases, it will be necessary that *we* decide how big the arrays and other data structures we use to hold strings are.

To do anything else with strings, we must typically call functions. The C library contains a few basic string manipulation functions, and to learn more about strings, we'll be looking at how these functions might be implemented.

Since C never lets us assign entire arrays, we use the `strcpy` function to copy one string to another:

```
#include <string.h>

char string1[] = "Hello, world!";
char string2[20];

strcpy(string2, string1);
```

The destination string is `strcpy`'s first argument, so that a call to `strcpy` mimics an assignment expression (with the destination on the left-hand side). Notice that we had to allocate `string2` big enough to hold the string that would be copied to it. Also, at the top of any source file where we're using the standard library's string-handling functions (such as `strcpy`) we must include the line

```
#include <string.h>
```

which contains external declarations for these functions.

Since C won't let us compare entire arrays, either, we must call a function to do that, too. The standard library's `strcmp` function compares two strings, and returns 0 if they are identical, or a negative number if the first string is alphabetically ``less than'' the second string, or a positive number if the first string is ``greater.'' (Roughly speaking, what it means for one string to be ``less than'' another is that it would come first in a dictionary or telephone book, although there are a few anomalies.) Here is an example:

```
char string3[] = "this is";
char string4[] = "a test";

if(strcmp(string3, string4) == 0)
```

```
                printf("strings are equal\n");
        else    printf("strings are different\n");
```

This code fragment will print ``strings are different''. Notice that `strcmp` does *not* return a Boolean, true/false, zero/nonzero answer, so it's not a good idea to write something like

```
        if(strcmp(string3, string4))
                ...
```

because it will behave backwards from what you might reasonably expect. (Nevertheless, if you start reading other people's code, you're likely to come across conditionals like `if(strcmp(a, b))` or even `if(!strcmp(a, b))`. The first does something if the strings are unequal; the second does something if they're equal. You can read these more easily if you pretend for a moment that `strcmp`'s name were `strdiff`, instead.)

Another standard library function is `strcat`, which concatenates strings. It does *not* concatenate two strings together and give you a third, new string; what it really does is append one string onto the end of another. (If it gave you a new string, it would have to allocate memory for it somewhere, and the standard library string functions generally never do that for you automatically.) Here's an example:

```
        char string5[20] = "Hello, ";
        char string6[] = "world!";

        printf("%s\n", string5);

        strcat(string5, string6);

        printf("%s\n", string5);
```

The first call to `printf` prints ``Hello, '', and the second one prints ``Hello, world!'', indicating that the contents of `string6` have been tacked on to the end of `string5`. Notice that we declared `string5` with extra space, to make room for the appended characters.

If you have a string and you want to know its length (perhaps so that you can check whether it will fit in some other array you've allocated for it), you can call `strlen`, which returns the length of the string (i.e. the number of characters in it), not including the `\0`:

```
        char string7[] = "abc";
        int len = strlen(string7);
        printf("%d\n", len);
```

Finally, you can print strings out with `printf` using the `%s` format specifier, as we've been doing in these examples already (e.g. `printf("%s\n", string5);`).

Since a string is just an array of characters, all of the string-handling functions we've just seen can be written quite simply, using no techniques more complicated than the ones we already know. In fact, it's quite instructive to look at how these functions might be implemented. Here is a version of `strcpy`:

```
mystrcpy(char dest[], char src[])
{
int i = 0;

while(src[i] != '\0')
        {
        dest[i] = src[i];
        i++;
        }

dest[i] = '\0';
}
```

We've called it `mystrcpy` instead of `strcpy` so that it won't clash with the version that's already in the standard library. Its operation is simple: it looks at characters in the `src` string one at a time, and as long as they're not `\0`, assigns them, one by one, to the corresponding positions in the `dest` string. When it's done, it terminates the `dest` string by appending a `\0`. (After exiting the `while` loop, `i` is guaranteed to have a value one greater than the subscript of the last character in `src`.) For comparison, here's a way of writing the same code, using a `for` loop:

```
for(i = 0; src[i] != '\0'; i++)
        dest[i] = src[i];

dest[i] = '\0';
```

Yet a third possibility is to move the test for the terminating `\0` character out of the `for` loop header and into the body of the loop, using an explicit `if` and `break` statement, so that we can perform the test after the assignment and therefore use the assignment inside the loop to copy the `\0` to `dest`, too:

```
for(i = 0; ; i++)
        {
        dest[i] = src[i];
        if(src[i] == '\0')
                break;
        }
```

(There are in fact many, many ways to write `strcpy`. Many programmers like to combine the assignment and test, using an expression like `(dest[i] = src[i]) != '\0'`. This is actually the same sort of combined operation as we used in our `getchar` loop in chapter 6.)

Here is a version of `strcmp`:

```
mystrcmp(char str1[], char str2[])
{
int i = 0;

while(1)
        {
        if(str1[i] != str2[i])
                return str1[i] - str2[i];
        if(str1[i] == '\0' || str2[i] == '\0')
                return 0;
        i++;
        }
}
```

Characters are compared one at a time. If two characters in one position differ, the strings are different, and we are supposed to return a value less than zero if the first string (`str1`) is alphabetically less than the second string. Since characters in C are represented by their numeric character set values, and since most reasonable character sets assign values to characters in alphabetical order, we can simply subtract the two differing characters from each other: the expression `str1[i] - str2[i]` will yield a negative result if the `i`'th character of `str1` is less than the corresponding character in `str2`. (As it turns out, this will behave a bit strangely when comparing upper- and lower-case letters, but it's the traditional approach, which the standard versions of `strcmp` tend to use.) If the characters are the same, we continue around the loop, *unless* the characters we just compared were (both) `\0`, in which case we've reached the end of both strings, and they were both equal. Notice that we used what may at first appear to be an infinite loop--the controlling expression is the constant 1, which is always true. What actually happens is that the loop runs until one of the two `return` statements breaks out of it (and the entire function). Note also that when one string is longer than the other, the first test will notice this (because one string will contain a real character at the `[i]` location, while the other will contain `\0`, and these are not equal) and the return value will be computed by subtracting the real character's value from 0, or vice versa. (Thus the shorter string will be

treated as ``less than'' the longer.)

Finally, here is a version of `strlen`:

```
int mystrlen(char str[])
{
int i;

for(i = 0; str[i] != '\0'; i++)
        {}

return i;
}
```

In this case, all we have to do is find the `\0` that terminates the string, and it turns out that the three control expressions of the `for` loop do all the work; there's nothing left to do in the body. Therefore, we use an empty pair of braces `{}` as the loop body. Equivalently, we could use a *null statement*, which is simply a semicolon:

```
for(i = 0; str[i] != '\0'; i++)
        ;
```

Empty loop bodies can be a bit startling at first, but they're not unheard of.

Everything we've looked at so far has come out of C's standard libraries. As one last example, let's write a `substr` function, for extracting a substring out of a larger string. We might call it like this:

```
        char string8[] = "this is a test";
        char string9[10];
        substr(string9, string8, 5, 4);
        printf("%s\n", string9);
```

The idea is that we'll extract a substring of length 4, starting at character 5 (0-based) of `string8`, and copy the substring to `string9`. Just as with `strcpy`, it's our responsibility to declare the destination string (`string9`) big enough. Here is an implementation of `substr`. Not surprisingly, it's quite similar to `strcpy`:

```
substr(char dest[], char src[], int offset, int len)
{
int i;
for(i = 0; i < len && src[offset + i] != '\0'; i++)
        dest[i] = src[i + offset];
dest[i] = '\0';
}
```

If you compare this code to the code for `mystrcpy`, you'll see that the only differences are that characters are fetched from `src[offset + i]` instead of `src[i]`, and that the loop stops when `len` characters have been copied (or when the `src` string runs out of characters, whichever comes first).

In this chapter, we've been careless about declaring the return types of the string functions, and (with the exception of `mystrlen`) they haven't returned values. The real string functions do return values, but they're of type ``pointer to character,'' which we haven't discussed yet.

When working with strings, it's important to keep firmly in mind the differences between characters and strings. We must also occasionally remember the way characters are represented, and about the relation between character values and integers.

As we have had several occasions to mention, a character is represented internally as a small integer, with a value depending on the character set in use. For example, we might find that `'A'` had the value 65, that `'a'` had the value 97, and that `'+'` had the value 43. (These are, in fact, the values in the ASCII character

set, which most computers use. However, you don't need to learn these values, because the vast majority of the time, you use character constants to refer to characters, and the compiler worries about the values for you. Using character constants in preference to raw numeric values also makes your programs more portable.)

As we may also have mentioned, there is a big difference between a character and a string, even a string which contains only one character (other than the `\0`). For example, `'A'` is *not* the same as `"A"`. To drive home this point, let's illustrate it with a few examples.

If you have a string:

```
char string[] = "hello, world!";
```

you can modify its first character by saying

```
string[0] = 'H';
```

(Of course, there's nothing magic about the first character; you can modify any character in the string in this way. Be aware, though, that it is not always safe to modify strings in-place like this; we'll say more about the modifiability of strings in a later chapter on pointers.) Since you're replacing a character, you want a character constant, `'H'`. It would *not* be right to write

```
string[0] = "H";                /* WRONG */
```

because `"H"` is a string (an array of characters), not a single character. (The destination of the assignment, `string[0]`, is a `char`, but the right-hand side is a string; these types don't match.)

On the other hand, when you need a string, you must use a string. To print a single newline, you could call

```
printf("\n");
```

It would *not* be correct to call

```
printf('\n');                   /* WRONG */
```

`printf` always wants a string as its first argument. (As one final example, `putchar` wants a single character, so `putchar('\n')` would be correct, and `putchar("\n")` would be incorrect.)

We must also remember the difference between strings and integers. If we treat the character `'1'` as an integer, perhaps by saying

```
int i = '1';
```

we will probably *not* get the value 1 in `i`; we'll get the value of the character `'1'` in the machine's character set. (In ASCII, it's 49.) When we do need to find the numeric value of a digit character (or to go the other way, to get the digit character with a particular value) we can make use of the fact that, in any character set used by C, the values for the digit characters, whatever they are, are contiguous. In other words, no matter what values `'0'` and `'1'` have, `'1' - '0'` will be 1 (and, obviously, `'0' - '0'` will be 0). So, for a variable `c` holding some digit character, the expression

```
c - '0'
```

gives us its value. (Similarly, for an integer value `i`, `i + '0'` gives us the corresponding digit character, as long as 0 <= `i` <= 9.)

Just as the character `'1'` is not the integer 1, the string `"123"` is not the integer 123. When we have a string of digits, we can convert it to the corresponding integer by calling the standard function `atoi`:

```
char string[] = "123";
int i = atoi(string);
int j = atoi("456");
```

Later we'll learn how to go in the other direction, to convert an integer into a string. (One way, as long as what you want to do is print the number out, is to call `printf`, using `%d` in the format string.)

Read sequentially: prev next up top