Lesson 9: C Strings

By Alex Allain

This lesson will discuss C-style strings, which you may have already seen in the array tutorial. In fact, C-style strings are really arrays of chars with a little bit of special sauce to indicate where the string ends. This tutorial will cover some of the tools available for working with strings--things like copying them, concatenating them, and getting their length.

What is a String?

Note that along with C-style strings, which are arrays, there are also string literals, such as "this". In reality, both of these string types are merely just collections of characters sitting next to each other in memory. The only difference is that you cannot modify string literals, whereas you can modify arrays. Functions that take a C-style string will be just as happy to accept string literals unless they modify the string (in which case your program will crash). Some things that might look like strings are not strings; in particular, a character enclosed in single quotes, like this, 'a', is not a string. It's a single character, which can be assigned to a specific location in a string, but which cannot be treated as a string. (Remember how arrays act like pointers when passed into functions? Characters don't, so if you pass a single character into a function, it won't work; the function is expecting a char*, not a char.)

To recap: strings are arrays of chars. String literals are words surrounded by double quotation marks.

```
"This is a static string"
```

Remember that special sauce mentioned above? Well, it turns out that C-style strings are always terminated with a null character, literally a '\0' character (with the value of 0), so to declare a string of 49 letters, you need to account for it by adding an extra character, so you would want to say:

```
char string[50];
```

This would declare a string with a length of 50 characters. Do not forget that arrays begin at zero, not 1 for the index number. In addition, we've accounted for the extra with a null character, literally a '\0' character. It's important to remember that there will be an extra character on the end on a string, just like there is always a period at the end of a sentence. Since this string terminator is unprintable, it is not counted as a letter, but it still takes up a space. Technically, in a fifty char array you could only hold 49 letters and one null character at the end to terminate the string.

Note that something like

```
char *my_string;
```

can also be used as a string. If you have read the tutorial on pointers, you can do something such as:

```
arry = malloc( sizeof(*arry) * 256 );
```

which allows you to access arry just as if it were an array. To free the memory you allocated, just use free:

For example:

```
free ( arry );
```

Using Strings

Strings are useful for holding all types of long input. If you want the user to input his or her name, you must use a string. Using scanf() to input a string works, but it will terminate the string after it reads the first space, and moreover, because scanf doesn't know how big the array is, it can lead to "buffer overflows" when the user inputs a string that is longer than the size of the string (which acts as an input "buffer").

There are several approaches to handling this problem, but probably the simplest and safest is to use the fgets function, which is declared in stdio.h.

The prototype for the fgets function is:

```
char *fgets (char *str, int size, FILE* file);
```

There are a few new things here. First of all, let's clear up the questions about that funky FILE* pointer. The reason this exists is because fgets is supposed to be able to read from any file on disk, not just from the user's keyboard (or other "standard input" device). For the time being, whenever we call fgets, we'll just pass in a variable called stdin, defined in stdio.h, which refers to "**st**an**d**ard **in**put". This effectively tells the program to read from the keyboard. The other two arguments to fgets, str and size, are simply the place to store the data read from the input and the size of the char*, str. Finally, fgets returns str whenever it successfully read from the input.

When fgets actually reads input from the user, it will read up to size - 1 characters and then place the null terminator after the last character it read. fgets will read input until it either has no more room to store the data or until the user hits enter. Notice that fgets may fill up the entire space allocated for str, but it will never return a non-null terminated string to you.

Let's look at an example of using fgets, and then we'll talk about some pitfalls to watch out for.

For a example:

```
#include <stdio.h>

int main()
{
    /* A nice long string */
    char string[256];

    printf( "Please enter a long string: " );

    /* notice stdin being passed in */
    fgets ( string, 256, stdin );

    printf( "You entered a very long string, %s", string );

    getchar();
}
```

Remember that you are actually passing the address of the array when you pass string because arrays do not require an address operator (&) to be used to pass their addresses, so the values in the array string are modified.

The one thing to watch out for when using fgets is that it will include the newline character ('\n') when it reads input unless there isn't room in the string to store it. This means that you may need to manually remove the input. One way to do this would be to search the string for a newline and then replace it with the null terminator. What would this look like? See if you can figure out a way to do it before looking below:

```
char input[256];
int i;

fgets( input, 256, stdin );

for ( i = 0; i < 256; i++ )
{
    if ( input[i] == '\n' )
    {
        input[i] = '\0';
        break;
    }
}
```

Here, we just loop through the input until we come to a newline, and when we do, we replace it with the null terminator. Notice that if the input is less than 256 characters long, the user must have hit enter, which would have included the newline character in the string! (By the way, aside from this example, there are other approaches to solving this problem that use functions from string.h.)

Manipulating C strings using string.h

string.h is a header file that contains many functions for manipulating strings. One of these is the string comparison function.

```
int strcmp ( const char *s1, const char *s2 );
```

strcmp will accept two strings. It will return an integer. This integer will either be:

```
Negative if s1 is less than s2.
Zero if s1 and s2 are equal.
Positive if s1 is greater than s2.
```

Strcmp performs a case sensitive comparison; if the strings are the same except for a difference in cAse, then they're countered as being different. Strcmp also passes the address of the character array to the function to allow it to be accessed.

```
char *strcat ( char *dest, const char *src );
```

strcat is short for "string concatenate"; concatenate is a fancy word that means to add to the end, or append. It adds the second string to the first string. It returns a pointer to the concatenated string. Beware this function; it assumes that dest is large enough to hold the entire contents of src as well as its own contents.

```
char *strcpy ( char *dest, const char *src );
```

strcpy is short for string copy, which means it copies the entire contents of src into dest. The contents of dest after strcpy will be exactly the same as src such that strcmp (dest, src) will return 0.

```
size_t strlen ( const char *s );
```

strlen will return the length of a string, minus the terminating character ('\0'). The size_t is nothing to worry about. Just treat it as an integer that cannot be negative, which is what it actually is. (The type size_t is just a way to indicate that the value is intended for use as a size of something.)

Here is a small program using many of the previously described functions:

```
#include <stdio.h>    /* stdin, printf, and fgets */
#include <string.h>   /* for all the new-fangled string functions */

/* this function is designed to remove the newline from the end of a string
entered using fgets.  Note that since we make this into its own function, we
could easily choose a better technique for removing the newline.  Aren't
functions great? */
void strip_newline( char *str, int size )
{
    int i;

    /* remove the null terminator */
    for (  i = 0; i < size; ++i )
    {
        if ( str[i] == '\n' )
        {
            str[i] = '\0';

            /* we're done, so just exit the function by returning */
            return;
        }
    }
    /* if we get all the way to here, there must not have been a newline! */
}

int main()
{
    char name[50];
    char lastname[50];
    char fullname[100]; /* Big enough to hold both name and lastname */

    printf( "Please enter your name: " );
    fgets( name, 50, stdin );

    /* see definition above */
    strip_newline( name, 50 );
```

```
    /* strcmp returns zero when the two strings are equal */
    if ( strcmp ( name, "Alex" ) == 0 )
    {
        printf( "That's my name too.\n" );
    }
    else
    {
        printf( "That's not my name.\n" );
    }
    // Find the length of your name
    printf( "Your name is %d letters long", strlen ( name ) );
    printf( "Enter your last name: " );
    fgets( lastname, 50, stdin );
    strip_newline( lastname, 50 );
    fullname[0] = '\0';
    /* strcat will look for the \0 and add the second string starting at
       that location */
    strcat( fullname, name );     /* Copy name into full name */
    strcat( fullname, " " );      /* Separate the names by a space */
    strcat( fullname, lastname ); /* Copy lastname onto the end of fullname */
    printf( "Your full name is %s\n",fullname );

    getchar();

    return 0;
}
```

Safe Programming

The above string functions all rely on the existence of a null terminator at the end of a string. This isn't always a safe bet. Moreover, some of them, noticeably strcat, rely on the fact that the destination string can hold the entire string being appended onto the end. Although it might seem like you'll never make that sort of mistake, historically, problems based on accidentally writing off the end of an array in a function like strcat, have been a major problem.

Fortunately, in their infinite wisdom, the designers of C have included functions designed to help you avoid these issues. Similar to the way that fgets takes the maximum number of characters that fit into the buffer, there are string functions that take an additional argument to indicate the length of the destination buffer. For instance, the strcpy function has an analogous strncpy function

```
char *strncpy ( char *dest, const char *src, size_t len );
```

which will only copy len bytes from src to dest (len should be less than the size of dest or the write could still go beyond the bounds of the array). Unfortunately, strncpy can lead to one niggling issue: it doesn't guarantee that dest will have a null terminator attached to it (this might happen if the string src is longer than dest). You can avoid this problem by using strlen to get the length of src and make sure it will fit in dest. Of course, if you were going to do that, then you probably don't need strncpy in the first place, right? Wrong. Now it forces you to pay attention to this issue, which is a big part of the battle.

Still not getting it? Ask an expert!
Quiz yourself
Previous: Arrays
Next: File I/O
Back to C Tutorial Index