# Chapter 8 - Strings and things

## Containers for strings

We have seen four types of values—characters, integers, floating-point numbers and strings—but only three types of variables—char, int and double. So far we have no way to store a string in a variable or perform operations on strings.

This chapter is going to rectify this situation and I can now tell you that strings in C are stored as an array of characters terminated by the character \0.

By now this explanation should make sense to you and you probably understand why we had to learn quite a bit about the working of the language before we could turn our attention towards string variables.

In the previous chapter we have seen that operations on arrays have only minimal support from the C language itself and we had to program extra functions by ourselves. Fortunately things are a little bit easier when we manipulate these special types of arrays - called strings. There exist a number of library functions in string.h that make string handling a bit easier than operations on pure arrays. Nevertheless string operations in C are still a lot more cumbersome than their equivalence in other programing languages and can be a potential source of errors in your programs, if not handled carefully.

## String variables

You can create a string variable as an array of characters in the following way:

char first[] = "Hello, ";

char second[] = "world.";

The first line creates an string and assigns it the string value "Hello." In the second line we declare a second string variable. Remember, the combined declaration and assignment is called initialization.

Initialisation time is the only time you can assign a value to a string directly (just as with arrays in general). The initialisation parameters are passed in the form of a string constant enclosed in quotation marks ("...").

Notice the difference in syntax for the initialisation of arrays and strings. If you like you can also initialize the string in the normal array syntax, although this looks a little odd and is not very convenient to type.

char first[] = {'H','e','l','l','o',',',' ','\0'};

There is no need to supply an array size when you are initialising the string variable at declaration time. The compiler compute the necessary array size to store the supplied string.

## Remember what we said about the nature of a string variable

It is an array of characters **plus** a marker that shows where our string ends: the termination character \0.

Normally you do not have to supply this termination character. The compiler understands our code and insertes it automatically. However, in the example above, we treated our string exactly like an array and in this case we have to insert the termination character ourselves.

When we are using a string variable to store different sting values during the lifetime of our program we have to declare a size big enough for the largest sequence of characters that we are going to store. We also have to make our string variable exactly one character longer than the text we are going to store, because of the necessary termination character.

We can output strings in the usual way using the printf() function:

printf("%s", first);

## Extracting characters from a string

Strings are called "strings" because they are made up of a sequence, or string, of characters. The first operation we are going to perform on a string is to extract one of the characters. C uses an index in square brackets ([ and ]) for this operation:

char fruit[] = "banana";

char letter = fruit[1];

printf ("%c\n", letter);

The expression fruit[1] indicates that I want character number 1 from the string named fruit. The result is stored in a char named letter. When I output the value of letter, I get a surprise:

a

a is not the first letter of "banana". Unless you are a computer scientist. For perverse reasons, computer scientists always start counting from zero. The 0th letter ("zeroeth") of "banana" is b. The 1th letter ("oneth") is a and the 2th ("twoeth") letter is n.

If you want the zereoth letter of a string, you have to put zero in the square brackets:

char letter = fruit[0];

## Length

To find the length of a string (the number of characters this string contains), we can use the strlen() function. The function is called using the string variable as an argument:

#include <string.h>

int main(void)

{

int length;

char fruit[] = "banana";

length = strlen(fruit);

return EXIT\_SUCCESS;

}

The return value of strlen() in this case is 6. We assign this value to the integer length for further use.

In order to compile this code, you need to include the header file for the string.h library. This library provides a number of useful functions for operations on strings. You should familiarize yourself with these functions because they can help you to solve your programming problems faster.

To find the last letter of a string, you might be tempted to try something like:

int length = strlen(fruit);

char last = fruit[length]; /\* WRONG!! \*/

That won't work. The reason is that fruit is still an array and there is no letter at the array index fruit[6] in "banana". Since we started counting at 0, the 6 letters are numbered from 0 to 5. To get the last character, you have to subtract 1 from length:

int length = strlen(fruit);

char last = fruit[length-1];

## Traversal

A common thing to do with a string is start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. A natural way to encode a traversal is with a while statement:

int index = 0;

while (index < strlen(fruit))

{

char letter = fruit[index];

printf("%c\n" , letter);

index = index + 1;

}

This loop traverses the string and outputs each letter on a line by itself. Notice that the condition is index < strlen(fruit), which means that when index is equal to the length of the string, the condition is false and the body of the loop is not executed. The last character we access is the one with the index strlen(fruit)-1.

The name of the loop variable is index. An **index** is a variable or value used to specify one member of an ordered set, in this case the set of characters in the string. The index indicates (hence the name) which one you want. The set has to be ordered so that each letter has an index and each index refers to a single character.

As an exercise, write a function that takes a string as an argument and that outputs the letters backwards, all on one line.

## Finding a character in a string

If we are looking for a letter in a string, we have to search through the string and detect the position where this letter occurs in the string. Here is an implementation of this function:

int LocateCharacter(char \*s, char c)

{

int i = 0;

while (i < strlen(s))

{

if (s[i] == c) return i;

i = i + 1;

}

return -1;

}

We have to pass the string as the first argument, the other argument is the character we are looking for. Our function returns the index of the first occurrence of the letter, or -1 if the letter is not contained in the string.

## Pointers and Addresses

When we look at the definition of the LocateCharacter() function you may notice the following construct char \*s which looks unfamiliar.

Remember, when we discussed how we had to pass an array to a function, back in Section~[Passing an array to a function](#Passing_an_array_to_a_function), we said that instead of copying the array, we only pass a reference to the function. Back then, we did not say exactly what this reference was.

C is one of the very few high-level programming languages that let you directly manipulate objects in the computer memory. In order to do this direct manipulation, we need to know the location of the object in memory: it's address. Adresses can be stored in variables of a special type. These variables that point to other objects in memory (such as variables, arrays and strings) are therefore called **pointer** variables.

A pointer references the memory location of an object and can be defined like this:

int \*i\_p;

This declaration looks similar to our earlier declarations, with one difference: the asterisk in front of the name. We have given this pointer the type int. The type specification has nothing to do with the pointer itself, but rather defines which object this pointer is supposed to reference (in this case an integer). This allows the compiler to do some type checking on, what would otherwise be, an anonymous reference.

A pointer all by itself is rather meaningless, we also need an object that this pointer is referencing:

int number = 5;

int \*i\_p;

This code-fragment defines an int variable and a pointer. We can use the "address-of" operator & to assign the memory location or **address** of our variable to the pointer.

i\_p = &number;

Pointer i\_p now references integer variable number. We can verify this using the "content-of" operator \*.

printf("%i\n", \*i\_p);

This prints 5, which happens to be the content of the memory location at our pointer reference.

With pointers we can directly manipulate memory locations:

\*i\_p = \*i\_p + 2;

printf("%i\n", number);

Our variable number now has the value 7 and we begin to understand how our LocateCharacter() function can directly access the values of string variables through the use of a char pointer.

Pointers are widely used in many C programs and we have only touched the surface of the topic. They can be immensely useful and efficient, however they can also be a potential source of problems when not used appropriately. For this reason not many programming languages support direct memory manipulation.

# String concatenation

In Section~[4.1](#Finding_a__character_in_a_string) we have seen how we could implement a search function that finds a character in a string.

One useful operation on strings is string **concatenation**. To concatenate means to join the two operands end to end. For example: shoe and maker becomes shoemaker.

Fortunately, we do not have to program all the necessary functions in C ourselves. The string.h library already provides several functions that we can invoke on strings.

We can use the library function strncat() to concatenate strings in C.

char fruit[20] = "banana";

char bakedGood[] = " nut bread";

strncat(fruit, bakedGood, 10);

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

The output of this program is banana nut bread.

When we are using library functions it is important to completely understand all the necessary arguments and to have a complete understanding of the working of the function.

The strncat() does not take the two strings, joins them together and produces a new combined string. It rather copies the content from the second argument into the first.

We therefore have to make sure that our first string is long enough to also hold the second string. We do this by defining the maximum capacity for string fruit to be 19 characters + 1 termination character (char fruit[20]). The third argument of strncat() specifies the number of characters that will be copied from the second into the first string.

# Assigning new values to string variables

So far we have seen how to initialise a string variable at declaration time. As with arrays in general, it is not legal to assign values directly to strings, because it is not possible to assign a value to an entire array.

fruit = "orange"; /\* Wrong: Cannot assign directly! \*/

In order to assign a new value to an existing string variable we have to use the strncpy() function. For example,

char greeting[15];

strncpy (greeting, "Hello, world!", 13);

copies 13 characters from the of the second argument string to the first argument string.

This works, but not quite as expected. The strncpy() function copies exactly 13 characters from the second argument string into the first argument string. And what happens to our string termination character \0?

It is **not** copied automatically. We need to change our copy statement to copy also the invisible 14th character at the end of the string:

strncpy (greeting, "Hello, world!", 14);

However, if we only copy parts of the second string into the first we need to explicitly set the n+1th character in the <

strncpy (greeting, "Hello, world!", 5); /\*only Hello is copied\*/

greeting[5] = '\0';

**Attention! In the last two sections we have used the strncpy() and the strncat() function that require you to explicitly supply the number of characters that will get copied or attached to the first argument string. The string.h library also defines the strcpy() and the strcat() functions that have no explicit bound on the number of characters that are copied. The usage of these functions is strongly discouraged! Their use has lead to a vast number of security problems with C programs. Remember, C does not check array boundaries and will continue copying characters into computer memory even past the length of the variable.**

## strings are not comparable

All the comparison operators that work on ints and doubles do work on strings. For example, if you write the following code to determine if two strings are equal:

if (word == "banana") /\* Wrong! \*/

This test will always fail.

You have to use the strcmp() function to compare two strings with each other. The function returns 0 if the two strings are identical, a negative value if the first string is 'alphabetically less' than the second (would be listed first in a dictionary) or a positive value if the second string is 'greater'.

Please notice, this return value is not the standard true/false result, where the return value 0 is interpreted as 'false'.

The strcmp() function is useful for putting words in alphabetical order.

if (strcmp(word, "banana") < 0)

{

printf( "Your word, %s, comes before banana.\n", word);

}

else if (strcmp(word, "banana") > 0)

{

printf( "Your word, %s, comes after banana.\n", word);

}

else

{

printf ("Yes, we have no bananas!\n");

}

You should be aware, though, that the strcmp() function does not handle upper and lower case letters the same way that people do. All the upper case letters come before all the lower case letters. As a result,

Your word, Zebra, comes before banana.

A common way to address this problem is to convert strings to a standard format, like all lower-case, before performing the comparison. The next sections explains how.

## Character classification

It is often useful to examine a character and test whether it is upper or lower case, or whether it is a character or a digit. C provides a library of functions that perform this kind of character classification. In order to use these functions, you have to include the header file ctype.h.

char letter = 'a';

if (isalpha(letter))

{

printf("The character %c is a letter.", letter);

}

The return value from isalpha() is an integer that is 0 if the argument is not a letter, and some non-zero value if it is.

It is legal to use this kind of integer in a conditional, as shown in the example. The value 0 is treated as false, and all non-zero values are treated as true.

Other character classification functions include isdigit(), which identifies the digits 0 through 9, and isspace(), which identifies all kinds of "white" space, including spaces, tabs, newlines, and a few others. There are also isupper() and islower(), which distinguish upper and lower case letters.

Finally, there are two functions that convert letters from one case to the other, called toupper() and tolower(). Both take a single character as an argument and return a (possibly converted) character.

char letter = 'a';

letter = toupper (letter);

printf("%c\n", letter);

The output of this code is A.

As an exercise, use the character classification and conversion library to write functions named StringToUpper() and StringToLower() that take a single string as a parameter, and that modify the string by converting all the letters to upper or lower case. The return type should be void.

## Getting user input

The programs we have written so far are pretty predictable; they do the same thing every time they run. Most of the time, though, we want programs that take input from the user and respond accordingly.

There are many ways to get input, including keyboard input, mouse movements and button clicks, as well as more exotic mechanisms like voice control and retinal scanning. In this text we will consider only keyboard input.

In the header file stdio.h, C defines a function named scanf() that handles input in much the same way that printf() handles output. We can use the following code to get an integer value from the user:

int x;

scanf("%i", &x);

The scanf() function causes the program to stop executing and wait for the user to type something. If the user types a valid integer, the program converts it into an integer value and stores it in x.

If the user types something other than an integer, C doesn't report an error, or anything sensible like that. Instead, the scanf() function returns and leaves the value in x unchanged.

Fortunately, there is a way to check and see if an input statement succeeds. The scanf() function returns the number of items that have been successfully read. This number will be 1 when the last input statement succeeded. If not, we know that some previous operation failed, and also that the next operation will fail.

Getting input from the user might look like this:

int main (void)

{

int success, x;

/\* prompt the user for input \*/

printf ("Enter an integer: \n");

/\* get input \*/

success = scanf("%i", &x);

/\* check and see if the input statement succeeded \*/

if (success == 1)

{

/\* print the value we got from the user \*/

printf ("Your input: %i\n", x);

return EXIT\_SUCCESS;

}

printf("That was not an integer.\n");

return EXIT\_FAILURE;

}

There is another potential pitfall connected with the scanf() function. Your program code might want to insist that the user types a valid integer, because this value is needed later on. In this case you might want to repeat the input statement in order to get a valid user input:

if (success != 1)

{

while (success != 1)

{

printf("That was not a number. Please try again:\n");

success = scanf("%i", &x);

}

}

Unfortunately this code leads into an endless loop. You probably ask yourself, why? The input from the keyboard is delivered to your program by the operating system, in something called an input buffer. A successful read operation automatically empties this buffer. However, if the scanf() function fails, like in our example, the buffer does not get emptied and the next scanf() operation re-reads the old value - you see the problem?

We need to empty the input buffer, before we can attempt to read the next input from the user. Since there is no standard way to do this, we will introduce our own code that reads and empties the buffer using the getchar() function. It run through a while-loop until there are no more characters left in the buffer (notice the construction of this loop, where all the operations are executed in the test condition):

char ch; /\* helper variable stores discarded chars\*/

while (success != 1)

{

printf("That isn't a number. Please try again:\n");

/\* now we empty the input buffer\*/

while ((ch = getchar()) != '\n' && ch != EOF);

success = scanf("%i", &x);

}

The scanf() function can also be used to input a string:

char name[80];

printf ("What is your name?");

scanf ("%s", name);

printf ("%s", name);

Again, we have to make sure our string variable is large enough to contain the complete user input. Notice the difference in the argument of the scanf() function when we are reading an integer or a string. The function requires a pointer to the variable where the input value will be stored. If we are reading an integer we need to use the address operator \& with the variable name. In the case of a string we simply provide the variable name.

Also notice, that the scanf() function only takes the first word of the input, and leaves the rest for the next input statement. So, if you run this program and type your full name, it will only output your first name.

## Glossary

**index:**

A variable or value used to select one of the members of an ordered set, like a character from a string.

**traverse:**

To iterate through all the elements of a set performing a similar operation on each.

**counter:**

A variable used to count something, usually initialized to zero and then incremented.

**concatenate:**

To join two operands end-to-end.

**pointer:**

A reference to an object in computer memory.

**address:**

The exact storage location of objects in memory.

Permission is granted to copy, distribute, transmit and adapt this work un- der the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 Inter- national License: https://creativecommons.org/licenses/by-nc/4.0/. If you are interested in distributing a commercial version of this work, please contact the author(s). The LATEX source and code for this book is available from: https://github.com/mkpenta/ThinkC