Course
ELEC-A7100
Course materials
Your points
Microsoft Teams
Code Vault

This course has already ended.

« 4. Strings Course materials 2 Functions for string handling »
ELEC-A7100 / 4. Strings / 1 Basics

Fourth round tasks as a `ZIP file.`

We will now look into **strings** in the C language. Actually strings are just an application of C arrays, with **char** - type array members (i.e., the individual characters). In the beginning you will learn to handle simple strings. The we step aside a for a bit, and take a look at few useful specifiers in C language, needed in understanding the string functions, that are discussed in the end of this section.

Basics¶

The C language does not have any particular string abstraction, but strings in C are just **char** type arrays that end in `'\0'` - character (`'\0'` represents numeric value 0). The final 0-charater is not visible, but it still uses one byte of memory in the end of the string. The ASCII value of the nil character is **0**.

Alternative ways to define a string¶

A simple example of a string is as follows:

```
char merkkijono[] = { 'Y','k','s','i',' ','j','u','t','t','u','\0' };
```

From here we can observe that a string is an array of *char* type values, which often are represented as character constants, as above. Because representing strings one character at a time is a bit tedious, strings can be initialised in an easier way as follows:

```
char merkkijono[] = "Yksi juttu";
```

The latter alternative does exactly the same thing as the first, but in a bit more easier way. The string constants are represented with double quotes. In this case we do not need an explicit nil character in the end, but the compiler adds it automatically at the end of the string. It is important to notice that in C **the double quotes and single quotes represent different things**: the former is for string constants, where as the latter signifies a single character constant. As generally with arrays, when the array or string is initialised at the same time with the variable declaration, the size of the table does not need to be explicitly indicated, because the compiler can determine this automatically.

If you need more space for a string (for example for future additions), you can specify the size explicitly:

```
char merkkijono[20] = "Yksi juttu";
```

A third way to specify a string is as follows:

```
const char *merkkijono2 = "Yksi juttu";
```

This differs from the previous alternatives in one significant way: unlike with the previous alternatives, the string pointed by *merkkijono2* **can not be modified** later in the program. This is because the string constants are placed in read-only section of the memory, alongside with the program code. To highlight this we have added a **const** specifier along with the variable declaration. It tells the compiler and the programmer that the string cannot be modified later on. Then the compiler can warn about incorrect use of the variable, which might terminate the program in a memory access exception. The *const* specifier is not mandatory, but recommended in this case to avoid programming errors. More about *const* specifier a bit later.

When a string variable is defined as an array, the string constant given with initialisation is copied as a local variable (as part of the call stack), in which case the string can be modified later.

The below diagram tries to illustrate this difference.

Outputing a string¶

Strings can be included in *printf* outputs by using the `%s` format specifier. One can supplement field size and other parameters also with this specifier, as with the others. Below is an example:

```
char string_B[] = "another string";
printf("My string is %s\n", string_B)
```

String and functions¶

In function arguments strings are usually represented using a *char* pointer, as with other kinds of arrays. Even if a string was originally declared as an array, it can be passed through a `char *` type argument. If the function is not going to modify the string, it is useful to specify the argument as `const char *`. Then the compiler knows that a string constant can be passed as a parameter.

In the following program there is function *clean_spaces* that walks through the given string and converts all spaces into period characters. In this function it would not be possible to specify the argument as *const*, because the function may modify the given string.

```
1  #include <stdio.h>
2
3  void clean_spaces(char *str)
4  {
5      while (*str != '\0') {  // Repeat until we are at the end of the string
6          if (*str == ' ') {  // is it a space?
7              *str = '.';  // replace it with a period
8          }
9          str++;  // move pointer to the next character
10     }
11 }
12
13 int main()
14 {
15     char message[] = "It is going to rain tomorrow";
16     clean_spaces(message);
17     printf("modified: %s\n", message);
18 }
```

In the above function one can see a typical example of who a string is processed using a pointer: intially the pointer points to the beginning of the string, after which the string will be processed in a *while* loop one character at a time. The loop ends when the pointer points the nil character that indicates the end of the string. Inside the loop the function tests whether the current charcter is a space, and if so, changes it to a period character. At the end of the loop the pointer is moved to the next character, after which the loop starts over.

In the above example it is useful to notice how the reference operator (`*`) is used. When we want to operate on a single character behind a pointer, we will use this operator (lines 5-7), that points to a single char value. When we want to change the pointer value, for advancing it forward (line 9), we will not use the reference operator. It may be difficult to realise when to use the reference operator in the beginning, but hopefully this will get clarified over the future exercises and examples.

Because in this example the content of the string is being modified, it is important that the string is declared as char array in the *main* function. You may try what happens if you change the specification on line 15 to the following form:

```
char *message = "It is going to rain tomorrow";
```

and then execute the program.

A string cannot be copied using an assignment operator. Instead you must copy the string one character at a time to its new location in the memory. There is a ready function *strcpy* available for this, even though it would be easy to do this using pointer arithmetics.

Task 4.1: Count alpha¶

Objective: Get familiar with operating on a string, character by character until the end of the string.

Write function `int count_isalpha(const char *str)` that counts the number of alphabetic characters in given string. You can use function **int isalpha (int character)** defined in `ctype.h` header, to check whether a single given character is alphabetic (i.e. you need to add a correct **#include** directive in the beginning of your source file). **isalpha** returns non-zero if the given character is alphabetic, or zero if it is not alphabetic. The `count_isalpha` function should return the number of alphabetic characters present in the given string.

Points **20 / 20** My submissions **2**
Deadline Friday, 2 July 2021, 19:59
To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Count Alpha

Select your files for grading

source.c

Choose File No file chosen

Submit

« 4. Strings Course materials 2 Functions for string handling »