C Programs: Strings

Strings in C

A **string** is a sequence of `characters`. `C` provides mechanisms for working with string constants and `string variables`.

String Constants

A string constant is any text enclosed in double quotes: ". Examples of string constants are:

```
"This is a sentence"
"a"
"25"
"-123+76"
"***___***"
```

String constants can be `#define`d in the program:

```
#define NAME "Alex"
#define MESSAGE "------> ERROR:"
#define BREAKS "\n\n\n"
```

We have seen use of string constants in `C` programs in `printf()` functions:

```
 printf("This is a sentence.");
 printf("Here, ends a line.\n");
 printf(BREAKS);
 printf(NAME);
```

will produce the following output:

```
This is a sentence. Here, ends a line.



Alex
```

String variables

While C has special syntax for string constants, it does not have a special string type.

Instead, C uses the definition above: *a string is a sequence of characters*, to represent string variables. C has an atomic type char for character values. A sequence of characters is **an array** of values of type char.

Thus, C uses character arrays to represent string variables.

Consider the following declarations:

```
char str[7];
char greeting[10] = "Hello!";
char alabama[10] = "Alabama";
char pacificStates[5][15] = {"California","Oregon","Washington", "Hawaii", "Alaska"};
```

These statements declare respectively: a string variable str of length 7 (characters), which does not get any value assigned to it; string variables greeting and alabama which get initial values assigned to them, and an array of strings pacificStates, which gets an initial assignment for each string in the array.

String I/O

String variables can be used in printf() and scanf() functions. A string value is indicated by a format string "%s" in the first parameter of both functions.

```
char name[20];
scanf("%s",name);
printf("%s\n", name);
```

Note: because name is an array, there is **no need** to use the & (address-of operator) in the scanf function.

String assignment

Wouldn't it be great if we could do this?

```
char str[10] = "Hello!";
char newStr[10];

newStr = str;
```

However, if you compile a program containing this code using -ansi -Wall -Werror settings, you will get:

```
In function main:
error: incompatible types in assignment
```

pointing to the newStr=str statement.

So, **how *do* we assign values to a string**?

Variant 1: A string is a character array. We can assign values to individual array elements. Consider, for example the following code fragment:

```
char str[10] = "Hello!";
```

```
char newStr[10];
int i;

for(i=0;i<10;i++) {
  newStr[i] = str[i];
}

printf("%s = %s \n\n",str,newStr);
```

This fragment declares and initializes to `"Hello!"` a string variable `str`. It then creates a string variable `newStr`, and in a `for` loop, assigns each element of `newStr` the value of the corresponding character from `str`.

You can always treat string variables declared as `char[]` as character arrays and work with this individual elements.

Variant 2: standard C library functions for managment of strings. Standard C library includes `string.h`, a header file that contains a variety of string management functions. Most of traditional operations that need to be performed on strings are covered in this library.

In the table below, the following new type constructs are used:

- `char * name`. As you know, `char * name` declares `name` to be a pointer to a character. This is a generic way to identifiy a string (sequence of characters) of *arbitrary length*.
- `const char *name`. This is used to specify that the parameter passed into a function is *input-only*. For example, `int foo(char * x, const char * y)` takes its first parameter `x` to be a `call-by-reference` modifiable parameter. It takes the second parameter to be a `call-by-value` non-modifiable one. As such, `foo(x,"boo");` is an acceptable function call for `foo`, whereas, it would not be acceptabe for the `int foo(char *x, char * y)`.
- `size_t`. Values of type `size_t` are *unsigned integers* (i.e., all be numbers that can be used to represent a size of a memory chunk).

Function declarataion	Meaning
`char * strcpy(char *dest, const char *source)`	copy the contents of `source` into `dest`
`char * strncpy(char *dest, const char *source, size_t n)`	copy forst `n` characters of `source` into `dest`.
`char * strcat(char *dest, const char * source)`	append `source` to end of `dest`
`char * strncat(char *dest, const char * source, int n)`	append up to n characters of `source` to end of `dest`
`int strcmp(const char *s1, const char *s1)`	compare two strings
`int strncmp(const char *s1, const char *s2, int n)`	compare first n characters of two strings
`size_t strlen(const char *s)`	determine the length of the string

To assign a value from string variable to another, use `strcpy()`.

String Comparison

Functions `strcmp()` and `strncmp()` are used to compare strings. The comparison follows the **lexicographical ordering** of strings.

Recall, that we can compare `char` values. E.g., `'c' < 'd'` returns `1` (true), while `'w' > 'x'` returns `0` (false).

Strings are arrays of characters, and they can be compared on character-by-character basis. Comparison starts with the first characters in strings and proceeds to the last characters. String `s` is **less than** string `t` if one of two conditions holds:

1. String `s` is equal to a prefix of string `t`, but the length of `t` is greater than the length of `s`.
2. String `s` starts with a prefix that is **less** than the prefix of `t` of the same size.

Examples. `"zone"` $<$ `"zones"`. Both strings start with the same prefix `"zone"`, but the second string has more characters following it.

`"table"` $<$ `"task"`. The first two characters of both strings, `"ta"`, coincide, however, the third character is different, and `'b'` $<$`'s'`, making `"tab"` $<$ `"tas"`. This implies that `"table"` $<$ `"task"`.

`"aqualang"` $<$ `"boost"`. While the length of the first string is greater than the length of the second string, the first characters of the two strings are `'a'` and `'b'` and `'a'`$<$`'b'`, implying the inequality.

Note: The order on the strings is essentially the same as the order in which words occur in a dictionary, except it extends to non-dictionary words, to strings containing other characters.

Task: Determine the order of the following strings: **"TABLE"**, `'table'`, `"Hello, World!"`, `"Hello World"`, `",."`, `".,"`.

strcmp(). C provides two functions, `strcmp()` and `strncmp()` to compare strings. Both functions, compare their input parameters and return the following:

- a negative number, if the first string is less than the second string;
- zero if both strings are equal;
- a positive number, if the first string is greater than the second string.

Consider the following code fragment:

```
char s[10] = "work";
char t[10] = "hard";

if (strcmp(s,t) < 0) {
    printf("Work hard\n");
}
else {
   if (strcmp(s,t) > 0) {
       printf("Hard work\n");
    }
    else {
       printf("hardwork\n");
    }
}
```

`"work"` is greater than `"hard"`, so, `strcmp()` returns a positive integer as the result, and therefore, this code fragment will print `"Hard work"`.

`strncmp()` works in a similar manner, only compares just the first n characters of two strings. E.g., `strncmp("move","movie", 3)` returns 0, while `strncmp("move","movie",4)` returns a negative integer.