Faculty of Computer Science, Dalhousie University *31-Oct-2018*

CSCI 2132 — Software Development

Lecture 24: Strings

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Pointers and arrays
- Pointer arithmetic: pointer + integer
- Pointer arithmetic: subtraction and comparison
- Pointer examples
- Mergesort revisited, with pointers

Mergesort2 Example

- to finish mergesort2 example

20 Strings

Strings are closely related to pointers and arrays in C. Now that we have learned pointers, let us learn about strings.

A C string is a sequence of characters followed by a null character (`'\0'`). Thus, a null character is used to mark the end of the string. A null character is a character whose numeric, i.e., ASCII, value is 0. This is why it can be written using the escape sequence '`\0`'. It is important not to confuse it with the character digit '`0`', which, by the way, has the ASCII value 48, i.e., octal value 60, so it could be writen using escape sequence '`\060`'.

When we count the length of a C string, we count the number of characters stored in this string, excluding the null character.

A C string can either be stored in a character array as a string variable, or it can be stored as a string literal.

20.1 String Literals

A string literal is a sequence of characters enclosed in double quotes. We actually saw a string literal in the first C program that we saw, which was:

```
"hello, world\n"
```

This string literal is stored in memory as a sequence of bytes, as follows:

h	e	l	l	o	,		w	o	r	l	d	\n	\0

or, in a different representation, these are actually the following numbers stored in memory: follows:

h	e	l	l	o	,		w	o	r	l	d	\n	\0
104	101	108	108	111	44	32	119	111	114	108	100	10	0

When using strings we frequently need to have in mind the byte required to store the null character at the end. Thus to store a string literal of n characters, we need $n + 1$ bytes.

We can assign string literals to character pointers. This makes the character pointer point to the starting position of the string. For example, we can write:

```
char *p = "hello, world\n";
```

Then, we can use operators for pointers that we learned before to access the characters in the string. Check the following statement:

```
char ch = p[0];
```

This statement will assign character 'h' to the variable `ch`. This is equivalent to:

```
char ch = *p;
```

20.2 String Variables

When allocating storage for a string variable, make sure to allocate an extra byte for the null character. In the following example, we define macro constant `STRLEN` to be the length of a string. Then when we declare a character array to store this string, we declare the array length to be `STRLEN+1`.

```
#define STRLEN 80
char str[STRLEN+1];
```

The actual length of the string that can be stored in `str` defined above can be anything from 0 to 80, since we can store a shorter string that does not require its full capacity.

Now, let us store the string `"abc"` in `str`. We can write:

```
str[0] = 'a';
str[1] = 'b';
str[2] = 'c';
str[3] = '\0';
```

However, this is quite tedious. Alternatively, we can use an initializer when we declare a string variable, as in:

```
char str1[6] = "abc";
```

Here, to store `"abc"`, we just need 4 bytes, including the null character at the end. The compiler will fill the remaining bytes with null characters, too. Thus, the following will be the content of `str1`:

a	b	c	\0	\0	\0

We can also omit the array length when we use an initializer:

```
char str2[] = "abc";
```

In this case, the variable `str2` will be an array of 4 elements, stored as:

a	b	c	\0

20.3 Reading and Writing Strings

We can use `printf` to print a string. The conversion specification for printing a string is `%s`. Thus, for the variable `str2` defined above, we can print it using:

```
printf("%s\n", str2);
```

We can also use another function in `stdio.h`. This is the function `puts`. If we call `puts(str)`, this will print the string `str` followed by a newline symbol. Of course, when printing a string, the delimiting null character is not printed, since it is not considered to be a part of the string.

To read a string, we can use `scanf`. There are several conversion specifications that can be used. The most common one is `%s`. For example, if the variable `str` is a character array, then we can write:

```
scanf("%s", str);
```

When we use `scanf` to read a string as shown above, `scanf` will first skip white spaces. It will then read characters and store them in `str`, until it encounters a white-space character, and finally it will store a null character to terminate the sting. For example, if the input is:

```
        hello, world
```

There are several space characters before `hello`, and a space character between comma and world. Then `scanf` will read and store the following string into `str`:

h	e	l	l	o	,	\0

This is how `scanf` can be used to read a word consisting of no-white-space characters. However, this use of `%s` conversion specifier is dangerous because we may come across a *very* long word, longer than the space we reserved to store it. A solution is to limit the size of the word that we can use by using the maximum field length, as in the following example:

```
char word[20];
scanf("%19s", word);
```

Even though we have a space of 20 characters in the word, we allow reading only 19 characters because the function `scanf` is going to add an additional null character at the end.