operating systems lab - week 3: exercise

Nicolo Colombo

November 3, 2020

This lab is about memory, pointers, arrays, and strings. You will see in practice how pointers and arrays are very similar objects and how to pass them to functions. You will learn how to handle strings by defining pointer arrays how to write C programs that accept command line arguments directly. Please try to complete all sections of this lab sheet before attempting this week's Moodle quiz

lab quiz - week 3

as some of the quiz questions may require you to run, comment or modify the programs you are asked to write.

Set up

Depending on your OS, use the following instructions to connect to `linux.cim.rhuk.ac.uk`:

Unix Open the terminal and run

```
ssh yyyyxxx@linux.cim.rhul.ac.uk
```

where `yyyyxxx` is your college username, and enter your password to access the teaching server.

Windows Launch the Windows SSH client `puTTY` [1], enter the following

```
linux.cim.rhul.ac.uk
```

in the empty field *Host Name (or IP address)* and click on *Open*. The client opens a new window where you are required to enter your college user name `yyyyxxx` and password.

Once logged in, you should be able to see the content of and navigate in your home directory using the standard UNIX commands, e.g. `ls`, `cd`, `cp`. Go to the `CS2850labs` directory[2] and run the command

```
$mkdir week3
```

to create a new sub-directory called `week3`. We suggest you save and compile all programs you write for this exercise in this directory.

Use a command-line text editor, e.g. emacs, nano or vim, to open, edit and save your programs. The advantage of command-line editors is that they can be used in a non-graphical SSH session. [3] You can create a new C file or open an existing C file, `file_name.c`, by running the command

```
$editorName file_name.c
```

We suggest you save separate files for all single parts of this exercise and follow the name suggestions given in each section. [4]

Compile your C code by running

```
$clang -o file_name file_name.c
```

and run the corresponding binary files `file_name` through

```
$./file_name
```

[1] `puTTY` should be installed on all department's machines. If you work on your own Windows machine you can download it at download puTTY and install it as explained.

[2] The parent directory you have created for the first week's lab

[3] If you do not want to open and close the editor every time you modify and save your code you can open a new SSH session and use two shell windows simultaneously.

[4] This is mainly because some of the Moodle quiz questions may refer to single pieces of code through the suggested file names.

For debugging, we suggest you use the free debugging tools of `valgrind`, which is already installed on the teacing server `linux.cim.rhul.ac.uk`. To check your code, you just need to run the command

```
$valgrind ./file_name
```

and have a look at the messages printed on the terminal.

1 Arrays

In this section, you will write a program that loads a set of integers entered by the user into an integer vector, prints all vector entries on separate lines and computes the vector squared norm using pointer arithmetics. The program input should be a series of nonnegative integers separated by spaces, e.g.

```
1 12 123 1234
```

In this case, a run of your program should produce the following output

```
cim-ts-node-03$ ./a.out
enter nonnegative integers:
1 12 123 1234
input: 1 12 123 1234
a[0] = 1
a[1] = 12
a[2] = 123
a[3] = 1234
<a,a> = 1538030
```

where the second line is the user's input. Save your code in a file called `array.c` and proceed as explained in the following subsections.

1.1 Parse the command-line input

Write a function,

```
int readLine(char *s, int MAX)
```

that reads the input character-by-character, loads the characters into a string, `s`, and stops reading if the number of characters reaches a fixed maximum value, `MAX`. Call `printf` in `main` to print the accepted input. Write and use `readLine` as suggested below.

- In `main`, define a fixed buffer size, e.g. `MAX = 100`, and declare `s` as

  ```
  char s[MAX];
  ```

- Use `getchar()` for reading the input characters from the terminal.
- Keep reading from the terminal until you encounter a new-line character, i.e. until the following condition is met

  ```
  (c = getchar()) == '\n'
  ```

  or you reach such a buffer size.
- Do not forget to null-terminate the string.
- For later use, make `readLine` return the number of characters loaded into `s`.

1.2 Load the vector entries

Write another function,

```
int loadVector(char *s, int len, int *a, int N)
```

that splits an input string, `s`, of length `len`, into a vector of integers, `a`, of maximum length `N`. Write and use `loadVector` as suggested below.

- In `main`, define a fixed vector length, e.g. `N` = 10, and declare the integer vector as

  `int a[N];`

  Note that`N` is the fixed maximum number of integers that can be loaded into `a` but `a` can have a fewer filled entries

- Make `loadVector` parse each block of integer characters, `0`, ..., `9`, by calling another function that converts a block of characters, e.g. `123`, into the corresponding integer. Let the conversion function be declared as

  `int getInt(char *s, int *pPos, int len)`

  where:

  i) `s` is the string of characters loaded by `readLine`,

  ii) `pPos` is a pointer to an `int`, `pos`, that keeps track of the position where the current block of characters starts, and

  iii) `len` is the length of the string returned by `readLine`.

  In particular, `getInt` should return the value that corresponds to the characters block starting at position `pos` and also update the position tracker.

- Try to understand why you should pass a pointer to `getInt` and not just an integer

- To avoid overflow problems, stop loading integers on `a` if `pos` exceeds the length of `s` or if the number of loaded integers gets greater than `N`. You can try to remove these limits, recompile your program and run it with a very long input to see what happens.

1.3 Compute the squared norm

The squared norm of a vector, v, is

$$\|v\|^2 = \sum_{i=1}^{|v|} v_i^2 \tag{1}$$

where $|v|$ is the length of v. To compute the squared norm of `a` in your program you can use the following function (after replacing `x` and `y` with the correct variable names):

```
long computeNorm(int *a, int len){
        long norm = 0;
        for(int n = 0; n<len; n++){
                norm = norm + *(x + y) * *(x + y);
        }
        return norm;
}
```

1.4 Print on the terminal

To print the entires of `a` in the required format, write a function

`void printVector(int *a, int len)`

that iteratively call `printf`. For practising with pointer arithmetics, try not to use the square brackets syntax, e.g. `a[2]`, to access the entries of `a`.

1.5 Test your program

To check your program, reduce the values of `MAX` and `N` and run it with long inputs. The program should only parse the allowed part of the input but behave correctly otherwise. Before doing that, run the executable with `valgrind` to see if you get error messages.

2 Strings

In this section, you will write a program that parses an input string, splits the input into words, stores the obtained words into a string array and prints single words on different lines. The program input should be a series of words separated by spaces, e.g.

```
one two three four five six seven eight
```

In this case, a run of your program should produce the following output

```
cim-ts-node-03$ ./a.out
enter words:
one two three four five six seven eight
input: one two three four five six seven eight (39)
w1: one (3)
w2: two (3)
w3: three (5)
w4: four (4)
w5: five (4)
w6: six (3)
w7: seven (5)
w8: eight (5)
```

In this section, `main` should *not* accept any command line arguments, i.e. it should start with

```
int main()
```

Save your code in a file called `strings.c` and proceed as explained in the following subsections.

2.1 Load the input

Define a maximum number of accepted characters at the beginning of `main`, e.g.

```
int MAXCHARS = 100;
```

and use `readLine` described in the previous section to store the command-line input into a string, `char s[MAXCHARS]`.

2.2 Compute the input length

Write a function

```
int stringLength(char *s)
```

that returns the number of characters of an input string `s`. Exploit the fact that the last character of a string is always `'\0'`.

2.3 Declare a pointer array

Set the maximum number of accepted words by adding

```
int MAXWORDS = 10;
```

at the beginning of `main` and declare an array of pointers to `char` of length `MAXWORDS`

```
char *t[MAXWORDS];
```

Each entry of `t` will point to a single word in `s`. Why is it important to specify the maximum number of words?

2.4 Split the input string

Define a function

```
int getWords(char *t[], char *s, int MAXWORDS)
```

that splits `s` where `s` contains an empty space, `' '`. Load the pointer to the beginning of each new word into an entry of `t`. Keep in mind that

- strings can be treated as pointers, i.e. if `char *q = "hello world"` than `q + 1` is `"ello world"`, `q + 2` is `"llo world"` , . . . , and
- strings are always null-terminated, i.e. if you replace the empty space in `q` by a null character, `'\0'`, all characters of `q` after the empty space will be cut out when you print it as a string.

For later use, make your function return the number of words found in `s`. Finally, to avoid overflow, use `stringLength` and the third argument of `getWords` to ensure that your function

- does not access memory slots that have not been reserved to `s`[5]
- does not load unreserved entires of `t`.

2.5 Print on the terminal

For printing `t`, use

```
void printVector(char **t, int size){
        for(int k = 0; k < size; k++){
                printf("w%d: %s (%d)\n", k + 1, *(t + k), stringLength(*(t+k)));
        }
}
```

where `size` is the integer returned by `getWords`. Try to understand the details of the pointer arithmetics in the input of `printf`.

2.6 Test your program

To check your program, reduce the values of `MAXWORDS` and `MAXCHARS` and run it with long inputs. The program should only parse the allowed part of the input but behave correctly otherwise. Before doing that, run the binary with `valgrind` to see if you get any error message.

3 Command line arguments

Write a new version of `string.c` where the input is passed directly to `main`, i.e. let `main` start with

```
int main(int argc, char *argv[])
```

Save your code into a new file called `arguments.c` and compare your program with the one of Section 2.6. What is the main difference between `argv` and `t`?

[5] The input may be shorter than `MAXCHARS`