UNIVERSITÉ DU
LUXEMBOURG

Programming 2

Christian Grévisse (christian.grevisse@uni.lu)

Lab 6 – Dynamic Memory Allocation & Strings

2018–03–29

Exercise 20 – Memory Allocation #malloc #MemoryManagement #Pointer #RandomNumbers

Write a C console application dynamically allocating memory. The program must at least implement the following functionalities:

1° ask the user for an integer value

- the value indicates the size of an array to be allocated, i.e. the number of integer values the array should be able to hold
- allocate such an array, fill it with random integer values and print the resulting array to standard output

2° Write a function that must be able to receive a string from the caller (in this case the `main()` function of the program)

- allocate an appropriate amount of memory to copy the string there twice
- return a pointer to this concatenated string to the caller
- print both the original and the concatenated string to standard output

Attention: If a function allocates memory for the caller, it is the caller's responsibility to free this memory as soon as it is no longer needed!

```
Enter a number:
100

30 91 15 72 61 41 10 37 98 41 94 80 26 96 10 88 59  5 84 14
26 13 83 54 87 41 46 37 53 19 60 51 54 78 94 76 91 56 60 94
62 80  7 37 66 90  7 12 58 26 26 40 14 36 45 14 99 45 87 20
46 18 97 75 12 28  3 71 96 11  3 29 12 52 31 83 92 22 68 65
53 73 76 94 91 76 91 79 61 43 18 63 74  3 29 50 36 59 69 95

Original: For a moment, nothing happened. Then, after a second or so, nothing continued to
    happen.

Double:   For a moment, nothing happened. Then, after a second or so, nothing continued to
    happen.For a moment, nothing happened. Then, after a second or so, nothing continued to
    happen.
```

Exercise 21 – Memory Allocation Loop #malloc #free

Write a C console application implementing the following task:

- two infinite loops, each allocating some memory every iteration
- the first loop must free the allocated memory every iteration
- whereas the second loop fails to free the allocated memory

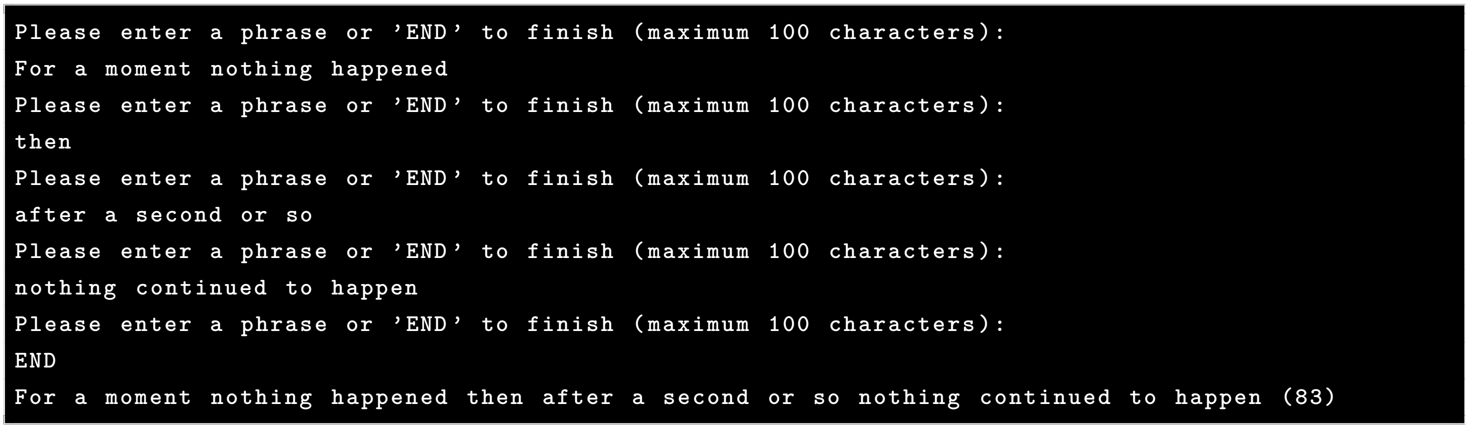
Prior to executing the loops individually, think about what will happen upon runtime. Then, use a tool like `top` to monitor.

Exercise 22 – Memory Reallocation #malloc #realloc #free

Write a C console application implementing the following task:

- get a string from the user via standard input

- store this string in a character array of exactly the necessary size
- get another string from the user via standard input
- add a blank space to the end of the character array followed by the newly acquired string reallocating only exactly the necessary amount of new memory
- repeat this procedure until the user enters "END" as input
- print the size and contents of the final array to standard output



```
Please enter a phrase or 'END' to finish (maximum 100 characters):
For a moment nothing happened
Please enter a phrase or 'END' to finish (maximum 100 characters):
then
Please enter a phrase or 'END' to finish (maximum 100 characters):
after a second or so
Please enter a phrase or 'END' to finish (maximum 100 characters):
nothing continued to happen
Please enter a phrase or 'END' to finish (maximum 100 characters):
END
For a moment nothing happened then after a second or so nothing continued to happen (83)
```

Exercise 23 – Memory Reallocation Outcome #malloc #realloc #free

Reallocating memory can have three outcomes: i) the reallocation fails (NULL is returned); ii) the reallocation works, and the pointer is still at the same place as before (i.e. the same pointer is returned); iii) the reallocation works, but the pointer has moved (i.e. a new pointer is returned).

Latter case is needed when the memory block of the size would not "fit" in the current location (more on this in Operating Systems 2).

Write a C console application that continuously reallocates a memory block of a bigger size each time. If the pointer has moved, print its old and new location on standard output.

Exercise 24 – Constant Strings #String #Pointer

The goal of this exercise is to recapitulate the difference between a constant string and a char array. Although arrays and pointers can be equally treated when referring to a variable (e.g. a function parameter), the way you *define* the variable is of utmost importance with respect to future actions.

1° Define a constant string, a char array and a pointer to latter. Does it make any difference when printing these strings

a) entirely;

b) character by character;

c) using the pointer syntax;

d) using the brackets syntax?

Why?

2° Modify the content of these variables. Which calls are valid?

3° What does the sizeof operator return on these 3 variables?

4° Using the strcpy function defined in string.h, what actions are allowed:

a) copy a constant string into a char array
b) copy a char array into a constant string
c) copy a char array into itself[1]

5° Define a constant string "Université".

a) Is there a difference between the results of `sizeof` and `strlen`? Explain!
b) What do you observe when comparing the number of characters returned by `strlen` and the actual number of letters? Explain! Hint: Print the hexadecimal representation of each character in the string and conclude.
c) Redo this question by defining the string as a char array. Do you observe any difference?

Exercise 25* – Address Sanitizer #BufferOverflow #DanglingPointer

Until now, you have probably used `gcc` (GNU Compiler Collection) to compile your C programs. But there exist alternatives, like `Clang`, a compiler front-end for a set of languages, backed by the LLVM compiler infrastructure project.

At Apple's WWDC (Worldwide Developers Conference) 2015, Anna Zaks presented the *Address Sanitizer*, a tool in the Xcode IDE that can detect, on runtime, several memory flaws, like dangling pointers and buffer overflows. The good news is that, while Xcode is only available on macOS, its Address Sanitizer tool actually uses the outcome of the Clang compiler, which in turn, *is* available also for other platforms, like Linux.

1° Write a C console application that, on purpose, contains flaws like a buffer overflow (e.g. caused by reading in a string with `scanf` but exceeding the available buffer size) and a dangling pointer (i.e. memory that has already been released but is used again).

2° On Linux, perform the following tasks in the command line[2]:

```
# Installation of Clang
sudo apt-get install clang

# Setup LLVM Symbolizer
sudo ln -s /usr/bin/llvm-symbolizer-3.8 /usr/bin/llvm-symbolizer
ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer

# Compile using Clang with adress sanitizing flag on
clang -O1 -g -fsanitize=address -fno-omit-frame-pointer main.c

# Execute
./a.out
```

What do you notice?

3° Watch the video from the WWDC to understand how Address Sanitizer works under the hood: WWDC 2015 – Session 413: Advanced Debugging and the Address Sanitizer. The talk on the Address Sanitizer starts at 13:45.

[1] The result of this operation may depend on your compiler and platform. For the version of `strcpy` as defined in the Linux kernel, refer to Linus Torvalds' GitHub repository

[2] More information: `http://clang.llvm.org/docs/AddressSanitizer.html`