

# **System Software**

## **C programming manual: lab 3**

### **2016 - 2017**

#### ***Bachelor Electronics/ICT***

*Course coördinator: Luc Vandeurzen*

*Lab coaches: Jeroen Van Aken*

*Stef Desmet*

*Tim stas*

*Luc Vandeurzen*

*Last update: January 26, 2017*

---

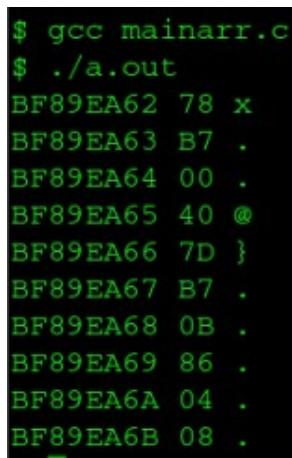
## C programming

*Lab targets: pointers in C, memory layout of a program*

*The following exercises illustrate step-by-step how the final result can be obtained: a small but powerful memory dump tool – called memdump - to print pieces of application memory.*

### **Exercise:** *pointers and memory access 1*

Implement a program that uses a pointer to visit every byte of a memory block to print the output as illustrated in the picture below.



```
$ gcc mainarr.c
$ ./a.out
BF89EA62 78 x
BF89EA63 B7 .
BF89EA64 00 .
BF89EA65 40 @
BF89EA66 7D }
BF89EA67 B7 .
BF89EA68 0B .
BF89EA69 86 .
BF89EA6A 04 .
BF89EA6B 08 .
```

The first column prints the memory address of every visited byte; the second column prints the value of this byte in hexadecimal format; and the third column prints the 'character' representation of the byte value if it's a printable char (library function!) and a '.' otherwise.

You can simply use an array to obtain a memory block:

```
typedef unsigned char byte_t;
#define DATA_SIZE 10
byte_t data[DATA_SIZE];
```

Hint: printing a memory address can be done with the '%p' format specifier.

### **Exercise:** *pointers and memory access 2*

As soon as the size of the memory block becomes bigger, printing info on every byte on separate lines results in ugly output. Inspiration for a much cleaner output can be found in tools such as Wireshark. Re-implement the previous exercise such that the output looks like the screenshot given below. In this example, a table is printed consisting of 3 columns and a number of rows depending on the memory size. The

first column contains the address of the first byte printed in the second column. The second column prints a sequence of bytes in hexadecimal format, starting from the address mentioned in the first column. The number of bytes that will be printed on 1 row should be set as a **constant** in the program. In the above example this constant is set to 10. The third column prints again all bytes from the second column but this time as characters. A '.' should be printed if a byte is not printable. Of course, it is possible that the last row is not a 'full' row anymore; spaces can be used to stuff that row. The first two lines show a simple but nice header for this table.

| Address  | Bytes |    |    |    |    |    |    |    |    |    | Chars         |
|----------|-------|----|----|----|----|----|----|----|----|----|---------------|
| BFF21A7C | 00    | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | .....         |
| BFF21A86 | 0A    | 0B | 0C | 0D | 0E | 0F | 10 | 11 | 12 | 13 | .....         |
| BFF21A90 | 14    | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D | .....         |
| BFF21A9A | 1E    | 1F | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | .. !"#\$\$%&' |
| BFF21AA4 | 28    | 29 | 2A | 2B | 2C | 2D | 2E | 2F | 30 | 31 | () *+, - ./01 |
| BFF21AAE | 32    | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 3A | 3B | 23456789:;    |
| BFF21AB8 | 3C    | 3D | 3E | 3F | 40 | 41 | 42 | 43 | 44 | 45 | <=>?@ABCDE    |
| BFF21AC2 | 46    | 47 | 48 | 49 | 4A | 4B | 4C | 4D | 4E | 4F | FGHIJKLMNO    |
| BFF21ACC | 50    | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | PQRSTUVWXYZ   |
| BFF21AD6 | 5A    | 5B | 5C | 5D | 5E | 5F | 60 | 61 | 62 | 63 | Z[\]^_`abc    |
| BFF21AE0 | 64    | 65 | 66 | 67 | 68 | 69 | 6A | 6B | 6C | 6D | defghijklm    |
| BFF21AEA | 6E    | 6F | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | nopqrstuvwxyz |
| BFF21AF4 | 78    | 79 | 7A | 7B | 7C | 7D | 7E | 7F | 80 | 81 | xyz{ }~...    |
| BFF21AFE | 82    | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 8A | 8B | .....         |
| BFF21B08 | 8C    | 8D | 8E | 8F | 90 | 91 | 92 | 93 | 94 | 95 | .....         |
| BFF21B12 | 96    | 97 | 98 | 99 | 9A | 9B | 9C | 9D | 9E | 9F | .....         |
| BFF21B1C | A0    | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | .....         |
| BFF21B26 | AA    | AB | AC | AD | AE | AF | B0 | B1 | B2 | B3 | .....         |
| BFF21B30 | B4    | B5 | B6 | B7 | B8 | B9 | BA | BB | BC | BD | .....         |
| BFF21B3A | BE    | BF | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 | .....         |
| BFF21B44 | C8    | C9 | CA | CB | CC | CD | CE | CF | D0 | D1 | .....         |
| BFF21B4E | D2    | D3 | D4 | D5 | D6 | D7 | D8 | D9 | DA | DB | .....         |
| BFF21B58 | DC    | DD | DE | DF | E0 | E1 | E2 | E3 | E4 | E5 | .....         |
| BFF21B62 | E6    | E7 | E8 | E9 | EA | EB | EC | ED | EE | EF | .....         |
| BFF21B6C | F0    | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | .....         |
| BFF21B76 | FA    | FB | FC | FD | FE | FF |    |    |    |    | .....         |

Hint: To avoid you spend a lot of time to get the formatting right, you can look at the following code that prints the first line of the header. You can find this code on Toledo.

```
printf("\n%-*s", ADDR_COLUMN_WIDTH, "Address");
```

```
printf("%s", COLUMN_SEPARATOR);  
printf("%-*s", BYTE_COLUMN_WIDTH, "Bytes");
```

A word of explanation on the used format specifiers might help. A format specifier like '%10s' indicates that a width of 10 characters should be used to print the argument. With '%-10s' you control not only the width but also the alignment of the argument: '-' left justifies the argument. Finally, '%\*s' controls the width value dynamically: the width value is given as an extra argument, being ADDR\_COLUMN\_WIDTH or BYTE\_COLUMN\_WIDTH in the above example. Notice that ADDR\_COLUMN\_WIDTH, BYTE\_COLUMN\_WIDTH and COLUMN\_SEPARATOR are nothing else then #define constants.

**Exercise:** *pointers and memory access 3*

**THE SOLUTION OF THIS EXERCISE NEEDS TO BE UPLOADED AS A ZIP FILE ON [syssoft.groep.tu.nl](https://syssoft.groep.tu.nl) BEFORE THE NEXT LAB.**

**YOUR SOLUTION IS ONLY ACCEPTED IF THE CRITERIA FOR THIS EXERCISE AS DESCRIBED ON [syssoft.groep.tu.nl](https://syssoft.groep.tu.nl) ARE SATISFIED!**

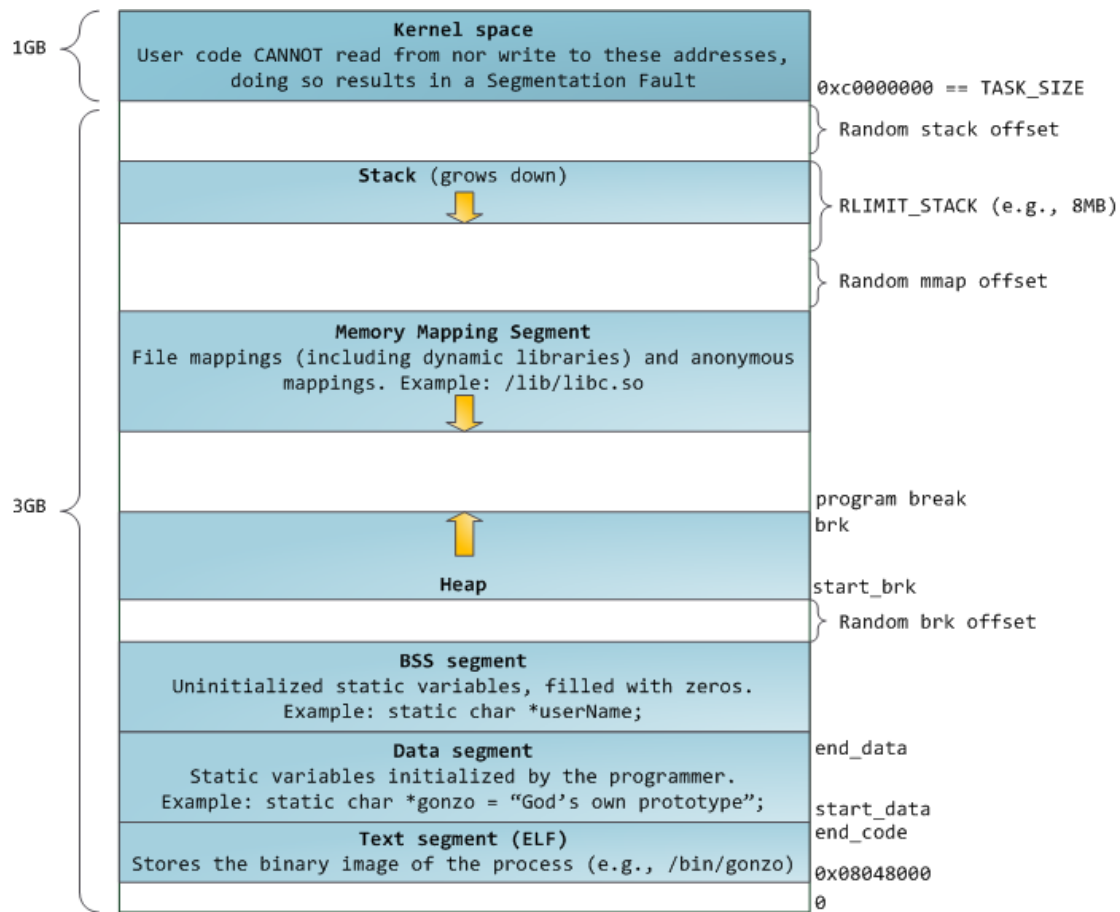
Hint: Use the code on Toledo as a starting point for this exercise.

And finally, we come to the implementation of the 'memdump' tool printing a real piece of application memory. Hence, you should no longer use the data array in the solution of the previous exercise. Instead, the tool reads the starting memory address and the memory size to print as input. The tool then prints out a memory dump in the format described in the previous exercise. Use a separate function to print out the memory dump such that you can easily re-use this tool in future exercises!

As an extra, we allow that the memory size input parameter can be a positive or negative number since you can move up or down in memory. Cleverly modify your code from the previous exercise to incorporate this feature.

Once finished, you can run your code for testing. But which memory address should you use to start with? You could try some address but most likely the program will crash with the message "segmentation fault". A famous message! A message that might be burned in your memory for the rest of your life after this course.

A Linux application runs in virtual memory these days. The mapping between virtual memory and physical memory is one of the concerns of Linux. If an application is started Linux assigns a block of virtual memory to this application. This block of memory starts at address 0x00000000 and is organized in logical sections such as .text, .data, .bss, heap and a function stack. A nice picture illustrating this is depicted below. Don't worry if you not fully understand this picture yet. At the end of this course, it should be more clear. Some memory regions are protected which means that if your program tries to access them, with a pointer for instance, Linux stops the program and tells you ... 'segmentation fault', indeed. This happens for instance if you try to dereference a NULL pointer, because NULL is typically defined as the 0x00000000 address which is part of a protected memory section.



The depicted figure shows the general memory layout of a program, but what are the actual address values of the function stack or .bss? One solution (there exist others as well) to show some real memory addresses is to access the `/proc` virtual file system. The Linux kernel uses `/proc` to make information on kernel data structures available. For each program currently running, the memory map can be accessed as the file `'/proc/<pid>/maps'`. More on processes and process IDs will follow later on in this course, but for now it suffices to know that the process ID identifies your running program. You can find the process IDs of running programs with the command `'ps'` or `'ps au'`. In C code, you can call the library function `'getpid()'` to obtain the process ID. Next you can print the correct `'/proc/<pid>/maps'` file with the `'cat'` command as follows:

```
#include <sys/types.h>
#include <unistd.h>
pid_t pid = getpid();
char *str;
asprintf(str, "cat /proc/%d/maps", pid);
system(str);
```

```
free(str);
```

The output of these instructions will look like:

```
str = cat /proc/5214/maps
08048000-08049000 r-xp 00000000 00:19 2249612      /home/u0066920/test/sysprog/lab2/exe
rcise3/a.out
08049000-0804a000 r--p 00000000 00:19 2249612      /home/u0066920/test/sysprog/lab2/exe
rcise3/a.out
0804a000-0804b000 rw-p 00001000 00:19 2249612      /home/u0066920/test/sysprog/lab2/exe
rcise3/a.out
b75cc000-b75cd000 rw-p 00000000 00:00 0
b75cd000-b7775000 r-xp 00000000 08:06 2229324      /lib/i386-linux-gnu/libc-2.19.so
b7775000-b7777000 r--p 001a8000 08:06 2229324      /lib/i386-linux-gnu/libc-2.19.so
b7777000-b7778000 rw-p 001aa000 08:06 2229324      /lib/i386-linux-gnu/libc-2.19.so
b7778000-b777b000 rw-p 00000000 00:00 0
b779a000-b779d000 rw-p 00000000 00:00 0
b779d000-b779e000 r-xp 00000000 00:00 0          [vdso]
b779e000-b77be000 r-xp 00000000 08:06 2229348      /lib/i386-linux-gnu/ld-2.19.so
b77be000-b77bf000 r--p 0001f000 08:06 2229348      /lib/i386-linux-gnu/ld-2.19.so
b77bf000-b77c0000 rw-p 00020000 08:06 2229348      /lib/i386-linux-gnu/ld-2.19.so
bfa52000-bfa73000 rw-p 00000000 00:00 0          [stack]
```

For this exercise, you can focus on the start and end addresses of the stack. You can go even one step further and add dummy local variables in the main function (or other functions as well) initialized to values easy to recognize, e.g. 'int var1 = 0xaaaaaaaa;' or 'int var4 = 0x11223344;'. Also print out the addresses of these variables so you can use them as memory anchor values with the memdump tool. A screenshot below illustrates the idea.

```
FYI: address of main function in memory: 80485FD
FYI: address of the first declared local variable of main(): BFDBEDD0
FYI: address of the second declared local variable of main(): BFDBEDD4
FYI: address of the third declared local variable of main(): BFDBEDD8
FYI: address of the last declared local variable of main(): BFDBEDE4

Enter start address (hex-notation) of dump: BFDBEDD0

Enter number of bytes to dump (negative or positive value): 20

Address  Bytes                               Chars
-----  -
BFDBEDD0 AA AA AA AA EE DD CC BB FF FF .....
BFDBEDDA FF FF 0A 00 00 00 D0 ED DB BF .....
$ █
```

### Exercise: *function stack*

Use the 'memdump' tool from the previous exercise to solve this exercise.

Write code to obtain more information on the organization of the function stack in memory, i.e. find answers on questions such as:

- Does the function stack grow 'up' or 'down' in memory?

- In which sequence are function arguments stored on the system stack (left to right or right to left)?
- What's the size on the stack of an array argument?

You can do this by calling functions with or without arguments, and with or without local variables. Store values in these variables that can be easily recognized, e.g.:

```
int a = 0xFFFFFFFF; // assuming 4 byte int
int b = 0xEEEEEEEE;
int c = 0xDDDDDDDD;
```