Lab 3: C Programming III

This laboratory requires the following equipment:

- C compiler
- GDB debugger

The laboratory duration is approximately 3 hours. Although this laboratory is not graded, we encourage you to take your own personal notes as the lab verification test might leverage results acquired during this laboratory session. For any questions, please contact us at **sis-ta@groupes.epfl.ch**

1.1 Information

In this assignment, you will find several exercises and questions.

- The notation $\mathbf{Q_x}$ means that the question can be answered theoretically or with simple commands in the Linux operating system, without implementing or running any code.
- The notation $\mathbf{S_x}$ means that the question can be solved only by compiling and running a piece of code or an additional simulation.
- The notation $\mathbf{I_x}$ means that the problem has to be solved by implementing, possibly compiling, and running a piece of code.
- The notation $\mathbf{B_x}$ means that the question is optional (bonus) and should be answered if you have enough time at your disposal.

1.2 Outline

This lab continues to exercise your C programming skills. In particular, you will practice pointers and memory management.

1.3 Getting Started (Short reminder)

To start with this lab, you will need to download the material available on Moodle. Download *lab03.tar.gz* in your personal directory. Now, extract the lab archive (you can type: `tar xvfz lab03.tar.gz`.)

1.4 Memory Management in C

S_1: The program below, when executed, prints "Your name is" followed by the name of the user taken as an input from the terminal. The source code is provided in the file `name_user.c`. Use `gcc` to compile your source file by typing the command `gcc name_user.c –o name_user` in the terminal. Then, execute the program.

```
#include <stdio.h>
void name_user(char* name) {
    printf("Your name is %s!\n", name);
}
int main(int argc, char* args[]) {
  char name[10];
  printf("Enter your name: ");
  scanf("%s", name);
  name_user(name);
  return 0;
}
```

Q2: Consider the case when the user's name is longer than 10 characters. What will happen then?

S3: Why? Now, re-execute the program and input a name much longer than 10 characters.

Q4: What if the user's name is exactly 10 characters long? *Hint: think about the termination character* `\0` *of a string.*

Q5: Can you think of a few problems that may arise as a consequence of exceeding an array bound?

B6: Why is it possible that the program from **S1** executes without an explicit warning when you enter a name equal to or slightly longer than 10 characters? In the general case, is it a safe practice to reserve a buffer which is a few characters shorter than needed for the input?

I7: You should always make an effort to make your program safe. By using a field of `scanf` function, you can limit the number of characters you will read from input to the size of your buffer. Consult the documentation of `scanf` and write a line of code that will read at most 9 characters.

Q8: You saw during the lecture that pointers and arrays are closely related. Compare the effect of the following two declarations on the memory and explain your reasoning.

```
char *p;
char a[20];
```

I9: Copy `name_user.c` to `name_user2.c`. Now, modify `name_user2.c` so that the user is asked to enter the size of the buffer (i.e., the length of his name), and allocate the memory for this buffer (using `malloc`). Do not forget to free the memory at the end (using `free`). *Hint: you will need to include* `<stdlib.h>` *library in order to use* `malloc` *and* `free`.

1.5 Data Structures

I10: In `account_structure.c` which you can find in the Code folder, a structure is defined to be used for account information. Complete the code in the `main()` function by assigning values to each field of the `struct` as you want and then print the contents. Use `gcc account_structure.c –o account_structure` for compiling and then execute.

Q11: Consider the definition of the `struct` in the previous question. The code is provided below. Notice, that there is one structure instance (`a`) and one pointer to the structure (`p`). How would you access the value of the account balance? Choose all answers that apply. You can make modifications to the code of **I10** and verify your answers.

```
struct account {
  int account_number;
  double balance;
  char *first_name;
  char *last_name;
} a;
struct account *p = &a;
```

a) `p.balance`

b) `a.balance`

c) `p->balance`

d) `a->balance`

e) `(&a).balance`

f) `*(p.balance)`

g) `(*p).balance`

h) `(&a)->balance`

1.6 Practice C on Paper

Q12: What is the result of this program? Compare your answers to the output you get when you run the program. **Justify your answer.**

```
#include <stdio.h>
int x = 2;
int y = 4;
void func(int y)
{
   int k=3;
   y = y+k;
   x = y*y;
}
void main(void)
{
  printf("%d %d \n",x,y);
   func(2);
   printf("%d %d \n",x,y);
}
```

Q13: Which statement is true after the following program fragment has been executed? **Justify your answer.**

```
    double number = 10.10;
    double *pt;
    pt = &number;
    *pt = 7.7;
```

a) `number` is 7.7 and `pt` is 7.7

b) `number` is 10.10 and `*pt` is 7.7

c) `number` is 10.10 and `*pt` is not known

d) `number` is 7.7 and `pt` is not known

1.7 Debugging using GDB

In this section, we want you to get familiar with the functionalities provided by gdb that allows for easier debugging. If you need to check the value of a variable, stop and check your variable if a certain condition is met, step-by-step go through your code to get a better understanding of what is going on or watch how a certain variable changes, there are already commands at your disposal. All you need to do is to learn how to use them and understand how to debug a program.

S14: Compile `test.c` with the `-g` flag using the command provided below and then type the second line in the command line. The first command causes `gcc` to store information in the executable program for gdb to make sense of it. The second command starts the debugger for testing your program. If you get a warning, read it carefully, it may help you in question **I16**.

```
gcc test.c -g -o test

gdb ./test
```

S15: Now we want to go through the lines of the program step by step. To do so you need to firstly set a breakpoint, then run your program in gdb and thirdly use the single-step command. To find these commands you can type help from within gdb. A selection of useful commands is given to you in the table below. After having tried a few single steps, try to run the rest of the program. This requires deleting or disabling breakpoints that the rest of your program may contain and continuing to run. What error do you get?

One way of doing so would be following the commands below. What does each command do? Why do you have to disable the breakpoint? Did you have to do this if the breakpoint was at line 12 of the code? Don't forget that you can also use other combinations of commands for debugging your program.

```
break lineNumber // for example 13

info break

run

step

continue

disable breakPointNumber

continue
```

I16: Try to debug the program so that it executes successfully. What were the problems and how did you solve them? You can use the commands below to see the variable contents.

```
print i

backtrace full
```

Is the value you get for `i` valid for the purpose of this code? Try to fix the first problem, recompile and start debugging again.

Now what error do you get? Try to check if the variables have valid values assigned to them. What is the value for `j`? What is `j` trying to do? Fix the code and try again.

Name	Arguments	Description
`start`		Starts the program.
`run`		Runs the program
`break`	Line number or function name	Sets a breakpoint in the given location
`step`		Single-step
`next`		Execute next line step over functions
`print`	expression	Displays the value of the variable
`watch`	expression	Sets a watchpoint for the expression
`continue`		Continue normal execution.
`Backtrace`	full	Shows the entire stack of the program
`delete/enable/disable`	Breakpoint or watchpoint	Delete/enable/disable
`Info`	args break watch	Show arguments of selected the frame show defined breakpoints show defined watch points
`quit`		Exit gdb

Hint: If you want more information on gdb commands you can have a look at http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf