# Lab 2: Advanced C (Part 1/2)

## 1. Preliminary Notes

- It was requested by students who have not used C in a long time that a review of that language should be given at the beginning of this course. It is, however, assumed that students have taken a C or C++ course in the past.
- The review will consist of three lectures and a class test. The lectures will be covered during lab time and will follow this document. An instructor (prof or TA) will lead the lab as a teaching and experiment-based environment. The problems will be solved on the (virtual) board.
- Lab 1 will review the basics of C, Lab 2 will review more advanced C, and Lab 3 will continue reviewing advanced C topics including memory allocation.
- **This lab is optional.** You do not need to attend the lab if you are comfortable with C.
- **The C test, however, is mandatory** and all students in this course must take the test.

## 2. Recursion

**PART 1: Compute N! recursively.**

Write a C program that computes N! (i.e., the factorial of a number N), *in a recursive way*. The program prompts the user for the integer number N, which is read from the keyboard. The result is returned to the `main` function and the result is displayed on the screen from the `main` function.

INPUT: An integer N, read from the keyboard.

OUTPUT: N! **(computed recursively)**

**PART 2: Program re-execution.**

Modify the program in PART 1 to ask the user whether or not to repeat the factorial computation, after the factorial is computed and displayed. If the user answers Y or y then the program loops back to the beginning and repeats everything. Any other input by the user terminates the program.

Example execution *(">" indicates reading waiting for input from the keyboard)*:

```
Enter N:
> 5
5! = 120
Continue? [Y/N]
> Y
Enter N:
> 3
3! = 6
> N
Bye!
```

## 2. Dynamic memory allocation

Dynamic memory allocation is an important feature of C. The fact that memory is handled directly by the programmer makes C a great fit for Operating Systems implementation, but it also makes it more difficult to write bug-free code. This exercise is meant to "flex" your C dynamic memory allocation muscle.

**PART 1: Pointers refresh**

Which of the three functions below can cause problems with pointers and memory allocation? Explain your answers for each of the functions.

```
A.  int* mystery_a(void){
      int n= 10;
      return (&n);
    }

B.  int * mystery_b(void){
      int* p;
      *p = 10;
      return p;
    }

C.  int* mystery_c(void){
      int *p;
      p = (int *) malloc (sizeof(int));
      *p= 10;
      return p;
    }
```

**PART 2: Freeing memory**

Is there a problem with the code snippet below? If yes, what is the problem? If no, explain why the code is correct.

```
void f(){
    int *p = (int *) malloc(sizeof(int));
    p = NULL;
    free(p);
}
```

**PART 3: What does this code do?**

Have a look at the code snippet below and, **without running it**, try to predict its output. Take a few minutes to think and write down the predicted output with an explanation for the program execution.

Now run the code. Did your prediction match the output?

If yes, congratulations! ☺ The exercise is done.

If not, re-visit the first step and try to understand why your prediction was not accurate.

```
# include<stdio.h>
# include<stdlib.h>

void my_int_malloc (int *n){
    n = (int*)malloc(sizeof(int));
}

int main(){
    int *p;
    my_int_malloc(p);
    *p = 7;
    printf("%d \n",*p);

    return(0);
}
```

## 3. Modular programming in C

The C programming language does not support object-oriented programming, but it supports modular programming. A module is a collection of functions that perform related tasks. For example, a module could exist to handle database functions such as lookup, enter, and sort. Another module could handle complex numbers, and so on. Each module has a well-defined interface that specifies the functions provided by this module. Moreover, each module has an implementation part that hides the code and any other private implementation detail the users of the module should not know/be concerned with.

**Public and Private.** Modules are divided into two parts: *public* and *private*.

- The **public** part tells the user how to call the functions in the module. It contains the definition of data structures and functions that are to be used outside the module. These definitions are put in a header file, and the file must be included in any program that depends on that module.
- Anything that is internal to the module is **private**. Everything that is not directly usable by the outside world should be kept private.

**Header and Source files.** In C, modular programming means splitting the source code of each module into a *.h header file* that specifies what that module exposes to the outside world, *and a corresponding .c implementation source file* where all the code and the details are hidden. The .h file has the same name as the .c file. They are viewed as a group. The .h header file contains only declarations of constants, types,

global variables and function prototypes that client programs are allowed to see and to use. Every other private variable or function internal to the module must stay inside the code file.

To take advantage of this private/public split, an application must be subdivided into individual .c files (modules) and *compiled individually* into .o object files. The .o files are generated by the compiler. Compiling a source file individually (without the other source files) gives the compiled code its own private named space.

To make certain functions private, the .c file is not shared with the project. **The developer only shares the .o compiled file and the .h header file.** Since .o files are compiled they are hard to read. The developer uses the .h file as a way to inform the rest of the project what is "public" in the .o file (declarations of constants, types, global variables and function prototypes). The developer can conveniently leave out functions that they do not want other to use, in this way, they have become "private".

In this exercise, you will take a program that does not respect the modular programming style, and modify it so that it becomes modular.

1. Open the provided source file `accounts-not-modular.c`
2. What does the program do?
3. Compile and run the program with `gcc -c accounts-not-modular.c`
4. Break up `accounts-not-modular.c`, so that it follows modular programming principles. Here are some suggestions:
   - create a separate `accounts.c` file, that contains the three functions related to account functioning, `deposit()`, `balance()`, and `accountNumberValid()`, and the accounts-related variables `balances[]` and `numberAttemptedTransactions`.
   - create a header file `accounts.h` that exposes the **accounts public interface**. Think about what variables and what functions need to be exposed. *Hint 1: One of the functions and one of the variables are helpers, or should stay private. Hint 2: You may need to use the* `extern` *keyword to expose public variables.* *https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/*

5. Rename your `accounts-not-modular.c` to `main.c` . At this point, your `main.c` should only contain the `main()` function and include the accounts header.

6. Compile and run your modular program:
   ```
   gcc -c accounts.c
   gcc -c main.c
   gcc main.o accounts.o
   ./a.out
   ```

   Notes: The output of `gcc -c` is a file with the same name of the .c file but with the file extension .o. We do this for both .c files. We do not do it for the .h file since it is included within the `main.c` file and gets compiled there. To merge the modules into a single application we write: `gcc main.o account.o`. We do not include the `account.h` file because it was already included within the `main.c` file. We then run the application by typing `./a.out`, which is the default name assigned to the application by gcc.