Warm-up sheet:
Programming in C

Programming for Embedded Systems
Uppsala University

January 20, 2015

Introduction

Here are some basic exercises in the programming language C. Hopefully you already have the required knowledge to solve them. They will not be graded and serve as a benchmark of programming skill that we expect you to have. Some general advice:

- Your code should be clear and easy to follow.
- Names of variables, functions and structures should make sense.

Introduction to Unix shell

You'll be working with Unix/Linux at some point in the course. If you are new to Unix or working from a terminal, the following table can be used as a quick reference. There are also many great resources online if you want to learn more.

`man <cmd>`	Shows a manual for the supplied command
`ls`	List files and directories in current directory
`pwd`	Show the working directory
`cd <dir>`	Go to the given directory
`cd ..`	Go back one directory
`mv <file1> <file2>`	Move file1 to file2
`rm <file>`	Remove a file
`rm -r <dir>`	Remove a directory and all files within it. Be careful, there is no undo button!
`cp <file1> <file2>`	Copy file1 to file2
`mkdir <dir>`	Create a directory
`Ctrl+C`	Key combination to abort most commands
`<command> &`	Run a command in the background
`gcc file.c -o output`	Compile `file.c` into executable file `output`
`emacs file.c &`	Open the editor emacs and load `file.c`
`./program`	Run the executable file `program` in the current dir

We will be using the GNU C Compiler (gcc). The manual for the compiler is quite long (15000+ lines), so don't worry about learning all of its features. To get you started, we go over the creation, compilation and execution of an example program.
First, we create a new directory to work in and step into that directory:

```
~]$ mkdir example
~]$ cd example
~/example]$
```

Then we create a C program file called `example.c`

```
~/example]$ emacs example.c
```

The file is edited to look as shown below:

```
/*
 * Author: Jonas Flodin
 */

#include <stdio.h>

int main(int argc, char ** argv){
  printf("This text is printed to the screen\n");
  return 0;
}
```

The file is saved by pressing **Ctrl+X** followed by **Ctrl+S**. Then we exit emacs by pressing **Ctrl+X** followed by **Ctrl+C**. (If you accidentally press some other command, you can abort it by pressing **Ctrl+G**). It should be noted that there are many editors other than emacs which you may prefer; vim, pico, gedit and nedit to name a few. You are of course welcome to use any editor of your choice.
We compile the program into an executable using gcc:

```
~/example]$ gcc -Wall example.c -o executable
```

The flag `-Wall` tells gcc to warn us about possible errors or design flaws that it can discover. It is a good habit to use this flag. You may also consider the `-std=c99` flag, which enables some newer additions to the C language, e.g., declaration of variables in the for loop header. Finally we list the files to see that some output has been produced and then we run the executable file.

```
~/example]$ ls
example.c  executable
~/example]$ ./executable
This text is printed to the screen
~/example]$
```

Now we've created and executed a program.

Exercises

Now that we know how to create and run a program, we move on to the exercises

Exercise 1 Input/output and conditionals

a) Write a program that outputs the string "Hello world!" to the terminal.

b) Have the program ask the user to input his/her name and then greet that name.

c) Ask the user to input his/her age. Then print the difference between your age and the input. The words "younger", "older" or "same age" should be used in the output, depending on the difference in age.

Example output:

```
[.../assignment1]$ ./a1e1
Hello world!
What is your name? Jonas
Hi Jonas!
How old are you? 80
You are 50 years older than me.
[.../assignment1]$ ./a1e1
Hello world!
What is your name? Maria
Hi Maria!
How old are you? 25
You are 5 years younger than me.
```

Exercise 2 Loops

a) Write a program that asks for a number. Then the program should print 1 through the given number on separate lines.

b) Encapsulate your code in a while-loop that asks the user if he/she would like to run the program again. Note that when reading a character from the input stream, the newline from the previous input is still buffered and considered as input. To discard the newline, start the *scanf* string with a space like this: `scanf(" %c", &input);`.

Example output:

```
[.../assignment1]$ ./a1e2
Give a number: 5
1
2
3
4
5
Run again (y/n)? y
Give a number: 2
1
2
Run again (y/n)? n
Exiting...
```

Exercise 3 Functions

a) Write functions for the four basic mathematical operations addition, subtraction, multiplication and division. Each function has two numbers as parameters and returns the result. Use integers. You do NOT have to do rounding for the division.

b) Write a program that asks the user for numbers a and b, and then use these numbers as arguments for your functions and print the result on the screen.

Example output:

```
[.../assignment1]$ ./a1e3
Give a: 11
Give b: 5
11 + 5 = 16
11 - 5 = 6
11 * 5 = 55
11 / 5 = 2
```

Exercise 4 Recursion

The Fibonacci sequence is a sequence of numbers where the first two numbers are 1 and 1 and the next number in the sequence is the sum of the previous two numbers. The n'th number in the sequence can be calculated as:
$f(1) = 1$
$f(2) = 1$
$f(n) = f(n-1) + f(n-2)$
See the example for the seven first numbers in the sequence.

a) Write a C function with a parameter n that returns the n'th Fibonacci number. The function must be recursive, i.e., it should call itself.

b) Write a program that asks the user for a number n and then prints the n first numbers in the Fibonacci sequence.

Example output:

```
[.../assignment2]$ ./a2e1
Give n: 7
1
1
2
3
5
8
13
[.../assignment2]$
```

Exercise 5 Arrays

In this exercise we look at some basic operations on arrays. Write a C function that:

a) counts the number of 0's in an integer array. The number of 0's is returned by the function.

b) prints an array of integers. The integers are printed on one line, enclosed in curly brackets and separated by a coma.

c) triples the value of all elements in an array of integers

All functions take two parameters, a pointer to the array of integers and the number of elements in the array.

Write a main function that declares and defines an array of integers, containing at least 10 integers. Use your other functions to print the initial array, the number of zero-valued elements in the array and the contents of the array when all elements have been tripled. Example output:

```
[.../assignment2]$ ./a2e2
Initial array: { 1, 2, 3, 0, 5, -6, 0, -8, 0, 10 }
Number of 0's: 3
Tripled array: { 3, 6, 9, 0, 15, -18, 0, -24, 0, 30 }
[.../assignment2]$
```

Exercise 6 Malloc and sorting strings

In this exercise we are going to implement an algorithm to alphabedically sort character strings. The algorithm we are using is called *bubble sort*, which is an easy, but inefficient sorting algorithm. The algorithm works as follows:

1 We are given an array of comparable elements.

2 We loop through the array, comparing the elements next to each other in the array. If a pair is in the wrong order, we swap their position.

3 If any position was changed, go back to step 2. If we didn't find a pair in the wrong order, we are done.

Our program is going to operate on an input of unkown size, which means that we have to do memory allocation dynamically using `malloc` and `free` from `stdlib.h`.

When comparing strings we use `strcmp` from the library `string.h`. The function `strcmp` considers the ASCII values of the characters when comparing, which means that numbers are considered smaller than uppercase letters which are smaller than lowercase letters.

Write a program that asks the user for how many strings to input, what the maximum string length is and then the actual strings. The program should then output the same strings in alphabetical order (according to `strcmp`). The program should be able to handle an arbitrary number of strings of an arbitrary maximum length. Make sure to free up your allocated memory.

Example output:

```
[.../assignment2]$ ./a2e3
Number of strings: 8
Maximum string length: 10
Give string 0: Hello
Give string 1: world!
Give string 2: Here
Give string 3: is
Give string 4: a
Give string 5: big
Give string 6: number:
```

```
Give string 7: 1234567890
Input when sorted:
1234567890
Hello
Here
a
big
is
number:
world!
[.../assignment2]$
```

Exercise 7 Simple array sorting

Write the function threeColorsSort that takes as input an array of integers in the range of 0 and 2 (0, 1 and 2 only), and arrange them in an increasing order:

```
void threeColorsSort(int * theArray, int arraySize)
```

To get full points, your solution should have linear runtime in the parameter `arraySize`.

Exercise 8 Strings

In this exercise, you will practice how to program with pointers and strings. Without using any library functions, write a C function

```
void append(char* str1, char* str2) { ... }
```

that takes as argument two strings `str1`, `str2` and appends `str2` to `str1`. After calling `append`, the pointer `str1` is supposed to point to the concatenation of (the original) `str1` and `str2`. The caller of `append` has to make sure that enough memory for the result of concatenation is available at the memory address that `str1` points to.

Example

```
char x[12] = { 'H', 'e', 'l', 'l', 'o', ' ',
               0, 1, 2, 3, 4, 5 };
char *y = "world";
append(x, y);
// now "x" contains the string "Hello world"
```

Your implementation needs to make sure that the output string (pointed to by `str1`) remains a well-formed string. Recall that, by definition, a string in C is an array of characters terminated with zero.

Is it possible that an invocation of `append` changes the string that `str2` points to? Argue why this is not possible, or give an example program where this happens. In the latter case, make sure that your implementation of `append` behaves in an acceptable manner also in such situations (e.g, your program is not supposed to end up in an infinite loop).

Exercise 9 Function pointers

The general understanding of a pointer is the memory address of some kind of data (integer, array, string, structure, etc ...).
A pointer can be as well pointing to a function.

Example Imagine we need to perform several kind of arithmetic operations on integers, let say: multiplying by two, resetting to zero, inverting the integer sign, etc ...
Therefore we define the following functions

```
void op_double(int * a) {...}
void op_reset(int * a) {...}
void op_invert(int * a) {...}
```

Example of use of one of those functions:

```
int a;
a = 5;
op_double(&a);
/* now (a == 10) holds */
```

We can now define a general function pattern using function pointer definition:

```
void (* arithmeticFuncPtr) (int *);
```

Notice that the star is related to the function, not to the returned value. If we consider an integer returning function, we would have the following:

```
int (* funcPtr) (int );
/* funcPtr is a pointer to a function that takes as argument
    an integer an returns an integer.*/

(int *) funcPtr (int );
/* funcPtr is a function that takes as argument
```

```
    an integer an returns a pointer to an integer.*/

(int *) (* funcPtr) (int );
/* funcPtr is a pointer to a function that takes as argument
    an integer an returns a pointer to an integer.*/
```

The following code illustrates use of the previously defined arithmetic pointer:

```
int a = 5;

// Assign the op_double function address to the
// function pointer.
arithmeticFuncPtr = &op_double;

// it also works to say: arithmeticFuncPtr = op_double

// Call of the op_double function using
// the arithmeticFuncPtr
(*arithmeticFuncPtr)(&a);

// it also works to say: arithmeticFuncPtr(&a)

// now (a == 10) is true
```

Thanks to their power, function pointers are used often in C libraries and frameworks; for instance, the FreeRTOS function `xTaskCreate` takes a function pointer as argument.

Here is what you need to do in this assignment:

1. Write the arithmetic functions `op_double`, `op_reset` and `op_invert`.
2. Write the function `applyTo` that takes as input a function pointer `func`, an array of integers `tab` and the array size `size`, and applies the pointed function to all the elements of the array:

   ```
   void applyTo(void (* func)(int *), int * tab, int size)
   ```

3. Use `applyTo` function to double the content of a 10 integers array.

Exercise 10 Command line arguments and file I/0

Write a program that outputs a multiplication table. The program takes 2 optional (for the user, not for you) arguments: input file and output file.

They are specified with `-in <filename>` and `-out <filename>`. The order should not matter. If no input file is specified, `stdio` is used as input. If no output file is `stdout` is used for output. The user specifies number of rows and columns, in that order, for the multiplication table with two integers. The columns must be aligned for all values not exceeding 1000. Remember to close any files that you opened.
Example output:

```
[.../assignment3]$ ./a3e1
4 5
   1    2    3    4    5
   2    4    6    8   10
   3    6    9   12   15
   4    8   12   16   20
[.../assignment3]$ ./a3e1 -in input.txt
   1    2    3    4    5    6
   2    4    6    8   10   12
   3    6    9   12   15   18
   4    8   12   16   20   24
   5   10   15   20   25   30
   6   12   18   24   30   36
   7   14   21   28   35   42
   8   16   24   32   40   48
   9   18   27   36   45   54
  10   20   30   40   50   60
  11   22   33   44   55   66
  12   24   36   48   60   72
[.../assignment3]$ cat input.txt
12 6
[.../assignment3]$ ./a3e1 -out table1.txt
2 3
[.../assignment3]$ cat table1.txt
   1    2    3
   2    4    6
[.../assignment3]$ ./a3e1 -out table2.txt -in input.txt
[.../assignment3]$ cat table2.txt
   1    2    3    4    5    6
   2    4    6    8   10   12
   3    6    9   12   15   18
   4    8   12   16   20   24
   5   10   15   20   25   30
   6   12   18   24   30   36
   7   14   21   28   35   42
   8   16   24   32   40   48
```

```
   9   18   27   36   45   54
  10   20   30   40   50   60
  11   22   33   44   55   66
  12   24   36   48   60   72
[.../assignment3]$
```

Exercise 11 Doubly linked list

Write a program where you declare a structure for a doubly linked list. The struct should contain a pointer to a string (name) as a key, a pointer to the previous element, a pointer to the next element and a birthdate. Please refer to the linked list C-code sent out by Kai for inspiration. Instantiate a list with input from `stdio`. End with inputing `Q` as name. Then output the list sorted alphabetically by name according to ASCII. Your solution should work for an arbitrary number of elements, with a maximum name length of at least 20 characters. Don't forget to free everything you allocate.
Example output:

```
[.../assignment3]$ ./a3e2
Name (Q to quit): Felix
Birthdate: 20121224
Name (Q to quit): Erica
Birthdate: 19980613
Name (Q to quit): Dawn
Birthdate: 19831004
Name (Q to quit): Charles
Birthdate: 19670225
Name (Q to quit): Benny
Birthdate: 20010810
Name (Q to quit): Astrid
Birthdate: 19901105
Name (Q to quit): Greg
Birthdate: 19940423
Name (Q to quit): Q
Astrid, 19901105
Benny, 20010810
Charles, 19670225
Dawn, 19831004
Erica, 19980613
Felix, 20121224
Greg, 19940423
[.../assignment3]$
```