HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2019
LAB 1: LINUX & C BASICS

The aim of this lab is to familiarise students with the the build process, to repeat and practice the fundamental concepts of the programming language C.

If you are not comfortable with the Linux/UNIX command line interface, please looks at the small tutorial located in the Student Portal under CONTENT/“Linux command line basics”.

Consult slides of Lecture 1 and Lecture 2 provided on Student Portal and search for information on the web.

Download the lab tar-ball `Lab01_Linux_and_C_Basics.tar.gz` from the Student Portal. Save it and unpack it.

Create a separate directory for each task and write the code for each task in the corresponding directory, for example

```
mkdir Task-1
cd Task-1
```

Part 1. Introduction

1. The build process

The process of transforming a written program into an executable in Linux/Unix takes 4 distinct steps: preprocessing, compilation, assembly and linking. See Lecture 1 slides for more details. For convenience, some or all of these steps can be performed with a single command.

Task 1:

Cd into the Task-1 directory. Compile the first program with the following command:

```
gcc -o first first.c
```

Here, we are using the gcc compiler. The ”-o first” flag tells the compiler to call the output file ”first”. The compiler then recognises the .c file ending, denoting a C source file, which it puts through the entire build process.

You can run the program by typing `./first`.

Date: January 21, 2019.

Now compile again giving another name, e.g. ”-o haha” instead of ”-o first”. Run the new “haha” and check that it works in the same way as “first”. You can also skip the -o flag and simply compile like this:

```
gcc first.c
```

What happens then? Use `ls` to see if any file was created. Can you run it? (The somewhat boring name a.out is usually the default executable name, if -o is not specified.)

2. Makefiles

When working with multiple compilation units (.c files), compiler flags, library paths, and so much more, it becomes necessary to make the build process repeatable and smooth. This is done with `make`. This program reads in and executes *makefiles*.

A makefile can contain a number of rules, defined like so:

```
target: dependency1 dependency2
<TAB>command-to-create-target
```

White space is important. There must be a TAB in front of the command. When `make` creates a target, it first looks at the specified dependencies. If the dependencies are files, then it only executes the command if any of those files have changed or the target is not present. If the dependencies are other targets, then it attempts to create those targets before continuing.

Task 2:

Cd into Task-2 and type `make` to build the `sorting` executable. By default, `make` will run the file called `makefile` (or `Makefile`) and build the first target (and therefore also any other targets that are dependencies of the first target). Type `make clean` to delete all the .o files in preparation for the next step.

Read `makefile` and understand how it works.

The `-w` flag turns off warnings. This is not good. Instead we want to turn on all warnings (`-Wall`), but you’ll notice that this makefile does not make this super-easy. Each time we want to change compiler flags we have to make changes to 3 or 4 lines of code.

The file `makefile-1` is more useful to us. By using variables to collect common elements in the commands, we can make changing the build process much more efficient.

Turn on all warnings by editing the `CFLAGS` variable in `makefile-1` .

Type `make -f makefile-1` to build the program again. (Do you get compiler warnings now?)

In practice, one usually does not write a makefile from scratch. We highly recommend that you save makefile-1 to an easy-to-remember directory and use it as a template in the future.

3. Debugging

Finding bugs in C programs in Linux without an IDE usually involves four things, presented roughly in order of decreasing importance (of course `-Wall` should be used as well):

(1) Good design. A careful program design with clearly defined and well documented/commented parts helps a lot.

(2) Using `assert` statements

(3) Using the `gdb` debugger

(4) Using `printf` statements

Task 3:

Cd into the Task-3 directory. In this task, the program is supposed to add the diagonal elements of a square matrix.

The program named `trace.c` consist of four functions:

- `initialization` (initialize the square matrix)
- `fill_vector` (fill a vector with random numbers from +10 to -10)
- `print_matrix` (display the matrix)
- `trace` (sum the diagonal values and return the sum)

Your task is to debug the program.

Turn on warnings. Or don't, if you want more of a challenge.

Use assert statements of the form `assert( bug_condition && "Description of error")` to trap for bugs.

For example, if at a certain point in the code you want to verify that the variable q has a positive value, you could use a line like this:

```
assert( q > 0 && "checking that q is positive");
```

or simply

```
assert( q > 0 );
```

Keep in mind that asserts are used to help finding programming errors during development, *not* for error-checking in a final release version of a code, so they are not the way to handle issues like bad user input.

All assert statements can be disabled by defining a macro with the name `NDEBUG` at some point before including assert.h. So, for a final release version of a program you can add a line like "`#define NDEBUG`" before assert.h is included and then all assert statements will be skipped by the compiler. This is a convenient way of making sure the release version of the code will not be slowed down by any assert statements.

Use GDB to locate segmentation fault bugs: First, compile with the `-g` flag to save debug info within the executable. Then, initialize the debugger with the `trace` executable inside by typing `gdb ./trace`. Type `run` or just `r` to start the program. `q` will exit gdb.

If you want to follow the program's execution with print statements, you can distribute a few `printf("%s: %d\n",__FILE__,__LINE__)` in the code using copy-and-paste.

Part 2. Programming in C

In this part you will write your own code for each given task, no external files from the Student Portal required for solving tasks are provided.

Note. You are not required to write makefiles in order to compile your code.

4. C basics

Task 4: Write a C program which reads two integer numbers a and b from the standard input using `scanf` function. Output using `printf` function and symbols '.' and '*' the rectangle of size $a \times b$. Example:

```
Input: 5 7
Output:
*******
*.....*
*.....*
*.....*
*******
```

Task 5:

A perfect square is an integer that is the square of an integer, for example the numbers 4 and 25 are perfect squares since $4 = 2^2$ and $25 = 5^2$. Write a C program which is checking if the number entered by the user is a perfect square. You can use the `sqrt` function from header `<math.h>`.

Note. The math functions in `<math.h>` have implementations in the library `libm.so`. If your program includes `<math.h>`, then you need to explicitly link to the math library:

```
gcc -o prog prog.c -lm
```

where `prog.c` is a name of your source file.

Task 6:

Modify your program from the previous task such that it accepts one argument passed from the command line. Your program will then check if the entered number is a perfect square. Make sure your program can be run like this:

```
./a.out 6
```

Check the number of input parameters and write an error message if a wrong number of input parameters is entered.

5. Pointers and memory allocation

Task 7:

Write a short C program that declares and initializes (to any value you like) a double, an int, and a char. Declare and initialize a pointer to each of the three variables. Output the value of each input variable, its address in hexadecimal format, and its memory size (in bytes).

Task 8:

The `main` function in a C program is the following:

```
int main()
{
int a,b;
char *s1,*s2;

a = 3; b=4;
swap_nums(&a,&b);
printf("a=%d, b=%d\n", a, b);

s1 = "second"; s2 = "first";
swap_pointers(&s1,&s2);
printf("s1=%s, s2=%s\n", s1, s2);
return 0;
}
```

The function `swap_nums` swaps values of two integers, and the function `swap_pointers` swaps values of two pointers. Write functions `swap_nums` and `swap_pointers` such that the program works and gives the following output:

```
a=4, b=3
s1=first, s2=second
```

Hint. Do not be afraid of the pointer to pointer notation! Think about the meaning of the following sentence: `char **x`.

Task 9:

Write a C program which reads a number n from the standard input and then reads n integer numbers into an array. Remove all prime numbers from the array putting the results in a new array. Output the elements of the new array and its size. The program should work for any number n, no maximum array size should be assumed.

Hint. Use `malloc` and `realloc` from the header `<stdlib.h>`.

Task 10:

Write a C program which reads integer numbers from the standard input into an array until a negative number appears. Then print all saved numbers and compute their sum. Note, the amount of numbers is not known, i.e. the length of the array is not known beforehand. The program should work for any number of input parameters, no maximum array size should be assumed.

Example:

```
Input: 4 5 6 7 34 5 -1 3 4 3 6
Output:
4 5 6 7 34 5
Sum: 61
```

Hint. Use `malloc` and `realloc` from the header `<stdlib.h>`.

6. File operations

Create a file `data.txt` with the following content:

```
5
Milk 10.3
Water 5.2
Potatoes 3.1
Carrots 4.8
Meat 20.0
```

The number written in the first line of the file is a total number of products. All following lines contain a name of a product and the corresponding price. The name of each product is a string of length not exceeding 50 characters.

Task 11:

Let product be represented in a code as a structure:

```
typedef struct product
{
char   name[50];
double price;
}
product_t;
```

Write a C program which reads data from a file and stores them into an array of products:

```
product_t *arr_of_prod;
```

Note that your program should not have any assumption on the number of products. The number of products is known only after opening and reading the first line in the file. When all data are read from the file, output data from the array to the standard output as a table.

7. Extra part

Look at the extra task if you are done with other tasks and have more time. If you need all your time for the non-extra tasks, then don't worry about the extra task.

Task 12:

Write a program that copies one file to another, converting lower case letters to upper case. Your program should accept both filenames as arguments passed from the command line. Use functions `fgetc`, `fputc` from the header `<stdio.h>` and `toupper` from the header `<ctype.h>`. For example if the input file is:

in.txt:

```
hello world
```

Then running

```
./a.out  in.txt out.txt
```

we get in out.txt:

```
HELLO WORLD
```