operating systems lab - week 1: exercise

nicolo colombo

This lab is about getting started with the environment for this course and practising the basic concepts of C. You will write your first programs in C and learn how to compile and execute them. The features of C that you will see in this lab, e.g. function calls, loops, and conditional statements, are very similar to other languages but you need to understand all the details of their syntax. You will write the code of a program that prints a fixed number of times the welcome string `hello world!`.

1 Getting started

1.1 Motivation

For pedagogical reasons, we ask you to

- use a command-line editor, e.g. emacs, vim, or nano,
- write, compile, and run your programs on the Linux environment provided by the department, i.e. the RHUL Computer Science *teaching server*, and
- connect to the teaching server using a command-line `ssh` client, e.g. `puTTY`

Working without a graphical interface may be annoying at the beginning, but you will be surprised by how quickly you will get used to it and the big gains in terms of stability, speed, and editing efficiency.

1.2 Connect to the teaching server `linux.cim.rhul.ac.uk`

Depending on your OS, use the following instructions to connect to `linux.cim.rhul.ac.uk`:

Unix Open the terminal and run

```
ssh yyyyxxx@linux.cim.rhul.ac.uk
```

where `yyyyxxx` is your college username, and enter your password to access the teaching server.

Windows Launch the Windows SSH client `puTTY` [1], enter `linux.cim.rhul.ac.uk` in the empty field *Host Name (or IP address)* and click on *Open*. The client opens a new window where you are required to enter your college user name `yyyyxxx` and password.

1.3 Create a new directory

You can now see the content and navigate in your home directory using the UNIX commands: `ls`, `cd`, `..` Create a new directory, called `CS2850Labs`, by running

```
mkdir CS2850Labs
```

You will use `CS2850Labs` to save and run all programs you will write for the CS2850 labs. Every week, you will create a sub-directory of `CS2850Labs` called `weekI`, `I` $= 1, \ldots, 11$, with the command

[1] `puTTY` should be installed on all department's machines. If you work on your own Windows machine you can download it at download puTTY and install it as explained.

```
mkdir weekI
```

Assuming that you start from your home directory, which contains `CS2850Labs` as a sub-directory, this week you should run

```
cd CS2850Labs
mkdir week1
cd week1
ls
```

Note that the last command is just to show the content of the current directory, i.e. `week1`, which, at the moment, should be empty. In general, if you need to see the content of a directory, e.g. `CS2850Labs`, and you are outside it, you can either run

```
ls directoryPath
```

where `directoryPath` is the directory's path, or move into it with `ls` and `cd`. You can always print on screen your current position in the tree by typing `pwd`. As an exercise, navigate back and forth between the new directories and your home folder.

1.4 Create and edit a text file

Choose one of the following command-line editors

- emacs
- nano
- vim

and have search in the online manual for the commands you need to
- open the editor and create a new text file
- open the editor and edit an existing file
- write and delete characters
- cut and copy characters, arbitrary strings or entire lines
- save the file and exit

When you feel familiar with the basic commands create a new file called `helloWorld.c`, type your name into it, save and exit. If you use vim, the series of commands you need is

```
vim helloWorld.c                                    1
i                                                   2
nicolo                                              3
ESC                                                 4
:w                                                  5
:q                                                  6
```

where the first command (Line 1) is a shell instruction that opens the editor and creates a new blank file called `helloWorld.c` and the others are Vim-commands for entering in the *edit mode* (Line 2), write your name (Line 3), exit the edit mode (Line 4), save (Line 5), and get you back to the terminal (Line 6). Finally, type

```
more helloWorld.c
```

to check that everything was saved correctly.

2 Your first C program

2.1 Write the C code

Open `helloWorld.c` again, replace your name with the following C code

```
#include <stdio.h>
int main() {
  printf("hello, world\n");
}
```

save, and exit.

2.2 Compile your C code

Compile `helloWorld.c` by running

```
gcc —Wall —Werror —Wpedantic helloWorld.c
```

To see all compilation options, type `man gcc` in the terminal and scroll the page with the up and down arrows. To read more about the meaning of the flags `-Wall`, `-Werror`, and `-Wpedantic` have a look at the gcc online manual on your web browser.

If you now print on the screen the content of `week1`, you should find a new file, `a.out`, which is the executable of `helloWorld.c`. Try to open it with vim. What do you observe? Why does the content of the file look so strange?

2.3 Run the executable

To see what `helloWorld.c` does, you need to execute the binary file, `a.out`, produced by the compiler. To do this, check that you are in the same directory as `a.out` and run

```
./a.out
```

Check that your output is exactly as follows

```
hello, world
```

2.4 More *hello, world*

Add the following lines to your code (just below the first call to `printf`)

```
printf("hello");
printf(",");
printf(" ");
printf("world\n");
printf("hello, world\n hello, world\n  hello, world!\n");
```

and check that the output is

```
hello, world
hello, world
hello, world
 hello, world
  hello, world!
```

including the strange indentation and the exclamation mark. What happens if you swap the exclamation mark and the last *newline* symbol, `\n`?

2.5 Debugging

The free system `valgrind` contains powerful debugging tools for Linux programs. The Valgrind suite is already installed on `linux.cim.rhul.ac.uk` and we suggest you use it to detect possible bugs in the programs you write for these labs. To see what may be wrong with your program, run the following

```
valgrind ./a.out
```

and have a look at the messages printed on the terminal. For the moment, this may look unnecessary and the messages you get are quite trivial. But running such sanity checks will become more and more important in the following weeks. One of the hardest parts of learning C is to understand how to manage the memory allocated by a program and looking at the `valgrind` messages may save you hours of debugging work.

2.6 The first C program of The C Programming Langauge

Kernighan and Ritchie's The C Programming Langaugeis the course reference book and you can find it online through the college library. What is the difference between the C code proposed in Section 1.1 of The C Programming Langaugeand the program shown in Section 2.1? Copy the code given in the book into a new file, `helloKernighan.c`, and try to compile by running

```
gcc —Wall —Werror —Wpedantic helloKernighan.c
```

What happens? Why no executable file is produced? Try again to compile `helloKernighan.c` without the additional flags `-Wall`, `-Werror`, and `-Wpedantic`. Why can you now see an executable file? How is it possible that the most famous book about C program contains an error in its very first line of code?

3 Control flow

The control structures `for` and `while` allows you to repeat an operation a given number of times. Their usage and syntax in C are similar to what you know from other programming languages, but we suggest you have a look at this C online manual for all the details.

3.1 Create loops with `for`

Copy the code given in Section 2.1 into a new file called `forHelloWorld.c`. Add the following *macro-substitution* instruction on Line 1

```
#define N 10
```

and write a `for`-loop to make the program print the string `hello, world` `N` times. See Section 4.11.2 of The C Programming Langaugefor more details about macro-substitution statements. A `for`-loop in C is specified by three quantities

- the *iterator*, which needs to be declared as an integer `int i` and initialised inside the `for`-loop arguments list
- the *stopping condition*, e.g. `i < 4`, which stops the iteration when *false*
- the *iteration step*, e.g. `i = i + 1`, which defines the increment of the iterator at each iteration

For example, a `for`-loop defined by

```
int i;
for (i = 3; i <= 6; i = i + 2) {
   doSomething(...);
}
```

will call the function `doSomething` 2 times.

3.2 `while`-loop (optional)

Have a look at the following program

```
#include <stdio.h>                                          1
#define N 10                                                2
int main() {                                                3
  int i;                                                    4
  int sum = 0;                                              5
  for (i = 0; i < N; i = i + 2) {                           6
    sum = sum + i;                                          7
    printf("%d + ", i);                                     8
  }                                                         9
  sum = sum + i;                                            10
  printf("%d = %d\n", i, sum);                              11
}                                                           12
```

Can you predict what is the output on the terminal without compiling and running the program? The following C code uses a `while`-loop and an `if`-statement to produce an analogous output

```c
#include <stdio.h>
#define N 10
int main() {
  int i;
  int sum = 0;
  while (i < N) {
    if (i % 2 == 0) {
      sum = sum + i;
      printf("%d + ", i);
    }
    i++;
  }
  sum = sum + i;
  printf("%d = %d\n", i, sum);
}
```

Write a new version of both programs, so that the output of both becomes

```
1 + 3 + 5 + 7 + 9 = 25
```

Use `Valgrind` to see if your program runs correctly and the heap usage of your program.