operating systems lab - week 2: exercise

Nicolo Colombo

October 2, 2020

This lab is about C types, variables and functions. You will see in practice how numbers and characters are represented in C, how you can define and call functions, and you will write programs that parse terminal input/output. Please try to complete all sections lab sheet before attempting this week's Moodle quiz

Lab quiz - week 2

as some of the quiz questions may require you to run, comment or modify the programs you are asked to write.

Set up

Depending on your OS, use the following instructions to connect to the teaching server, `linux.cim.rhuk.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 the following

  `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.

Once logged in, you should be able to see the content of and navigate in your home directory using the standard UNIX commands, e.g. `ls`, `cd`, `cp`. Go to the `CS2850labs` directory[2] and run the command

`mkdir week2`

to create a new sub-directory called `week2`. We suggest you save and compile all programs you write for this exercise in this directory.

Use a command-line text editor, e.g. emacs, nano or vim, to open, edit and save your programs. The advantage of command-line editors is that they can be used in a non-graphical SSH session. [3] You can create a new C file or open an existing C file, `file_name.c`, by running the command

`editorName file_name.c`

We suggest you save separate files for all single parts of this exercise and follow the name suggestions given in each section. [4]

Compile your C code by running

`clang -o file_name file_name.c`

and run the corresponding binary files `file_name` through

`./file_name`

[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.

[2] The parent directory you have created for the first week's lab

[3] If you do not want to open and close the editor every time you modify and save your code you can open a new SSH session and use two shell windows simultaneously.

[4] This is mainly because some of the Moodle quiz questions may refer to single pieces of code through the suggested file names.

For debugging, we suggest you use the free debugging tools of `valgrind`, which is already installed on the teacing server `linux.cim.rhul.ac.uk`. To check your code, you just need to run the command

```
valgrind ./file_name
```

and have a look at the messages printed on the terminal.

1 Types

In this section you will be experimenting with variables of different types and see how type conversion works in practice.

1.1 `sizeof`

Write a program, `sizeOfTypes.c`, that prints on the terminal the size in bytes of the following types:

```
void, char, int, unsigned int, float
```

The size of a given type can be obtained by calling the function `sizeof(type)`. The program should print the size of each type on a different line, with each line being of the form

```
the size of a long int is 8 bytes
```

Note that the output of `sizeof` is an `unsigned long int`.

You can also use `sizeof` to obtain the size of pre-declared variables, e.g.

```
int a;
unsigned long int bytes = sizeof(a);
```

save into `bytes` the size of `a`. What happens if you use `sizeof` to get the memory size associated with an array? Modify your program so that it prints two extra lines reporting the size in bytes of a 10-dimensional array of `char` and `int` declared as

```
int vInt[10];
char vChar[10];
```

1.2 Signed or unsigned `char`

The conversion of characters to integers depends on whether the variables of type `char` are signed or unsigned quantities. Write a simple program, `signOfChar.c`, to understand if on `linux.cim.rhul.ac.uk` they are signed or unsigned.

Suggestion Try to understand why `clang` produces an error message if your program includes a declaration such as

```
char c = 150;
```

Try to compile the same program using `gcc`.

1.3 `int`, `char` and `unsigned int`

The conversion of a variable of type `int` into type `char` may cause some information to be lost. Copy the following program

```
#include <stdio.h>
int main(){
        char c;
        int i = 123;
        c = i;
        i = c;
        printf("i=%d\n", i);
}
```

into a new file, `intToChar.c`. Can you explain what happens if you change `int i = 123;` with `int i = -123` or `int i = 130` and `int i = - 130`?

1.4 Operation with unsigned integers

Copy the content of `intToChar.c` into a new file called `intToUnsigned.c`. In `intToUnsigned.c`, modify your program by replacing

```
char c;
```

with:

1. `unsigned int c;`
2. `unsigned short c;`

and repeat the experiments of Section 1.3 Can you explain the output you obtain in both cases?

2 `getchar` and `putchar`

In this section, you will write a program that transofrms all lower case letters of an input string into upper case letters. We suggest you follow the following steps:

- The standard library contains functions for reading or writing one character at a time are:
  1. `getchar()`, that reads the next input character and returns it, and
  2. `putchar(c)`, that prints the character `c` on the terminal.

  Copy the following code

  ```
  #include <stdio.h>
  int main(){
          int c;
          while ((c = getchar()) != EOF){
                  putchar(c);
          }
  }
  ```

  into a new file, `ascii.c`, compile it and try to understand how it works. When you run the program the terminal shows a new empty line where you can type your text. To exit you need to send an `EOF` signal to the program, which you can do by typing `ctrl-d`. [5]

- Write a function,

  `int upper(int c) ...`

  that check if the input character, `int c`, is a lower case letter and in that case tranform it into the corresponding upper case letter. `upper` can be a modified version [6]

  ```
  int lower(int c){
      if (c >= 'A' && c <= 'Z')
          return c + 'a' - 'A';
      else
          return c;
  }
  ```

[5] Note that `c` is declared as an `int` to include `EOF`, which is an integer but can not represented as a `char`.

[6] `lower` works only for ASCII characters because upper case and lower case letters are at a fixed distance. The library `<ctype.h>`, see Appendix B the Kernighan and Ritchie's book, provides analogous functions that work independently from the character set.

- Call `lower` from the loop of `main` given above to transform the characters that you read from the terminal.
- In the same loop, call `putchar(c)` to print the obtained upper case letters.
- Copy the following text

  ```
  one two three
  four five
  six
  ```

  into a file, `someText.txt`, and see what happens when you run the following command

  ```
  ./ascii < someText.txt
  ```

  Keep a working copy of `ascii.c` because you may be asked to run it to answer some questions of this week's Moodle quiz.
- Instead of printing the output of `upper(c)` directly on the terminal you can save the outputs of `upper(c)` into a string, i.e. an array of `char`, [7] of length $n = 10$. In your program, declare the string as

  ```
  int n = 10;
  char string[n];
  ```

  and print `string` on the terminal before exiting. A string can be printed using

  ```
  printf("%s", string);
  ```

  where `% s` is the format specifier for printing a string. [8] Write new version of `ascii.c` where you implement the above and save your code in a new file, `asciiString`.
- Test `asciiString.c` with both command-line input and file input. Try also to vary the values of `n`. What do you observe? Be sure that your program does not behave strangely when the input is longer than the value of `n`. After running some sanity checks, change the value of `n` to 100 so that `asciiString.c` prints the upper-case version of all lines in `someText.txt`.

2.1 Binary representation (optional)

Write a new program, `binary.c`, that:

1. reads characters from the terminal (through `getchar`) until the user sends a new line character;[9]
2. converts the first n *numerical characters* of the input [10] into the corresponding integers and save them into an n-dimensional integer array declared as

   ```
   int array[n];
   ```

3. computes the decimal integer obtained by grouping the entries of `array` into a single integer and save it into an integer variable, `val`. You can use the following formula

$$N = \sum_{i=1}^{n} s[i] * 10^{n-i}$$

[7] Remember that the last character of a string should be `'\0'`.
[8] Other useful format specifier are:
`%c` - - to print value in character format
`%u` - - to print value in unsigned integer format
`%d` - to print value in integer format.
`%o` - to print value in octal format.
`%x` - to print value in hexadecimal format (letters will print in lowercase)
[9] In the terminal this is done by pressing `enter`.
[10] A numerical character is a character in the set $\{0, 1, ..., 9\}$

where s is a vector of n single-digit integers, e.g. $N = 123$ when $n = 3$ and $s = [1, 2, 3]$. The program should ignore all non-integer characters, i.e. the value of `val` should be the same if the user's input is `1one2two3three`, `12and3` or `123`

4. extracts the binary representation of `val` and prints it on the terminal.

Hint The binary representation of an unsigned integer can be obtained by calling the following function

```
int getbits(unsigned x, int p){
        int mask = ~((unsigned) ~0 << 1);
        x = x >> p;
        x = x & mask;
        return x;
}
```

which extracts the `p`-th bit of the binary representation of `x`. Before copying `getbits` into your program, have a look at how the bitwise operators, `>>`, `<<`, `&` and , work.[11] To check the output of your program you can use `printf` with the `%o` format specifier (that prints the octal form of the input). [12]

[11] For example, in Section 2.9 of the Kernighan and Ritchie's book.

[12] You can swiftly compare octal and binary representations by looking at the following table:

0	- 000
1	- 001
2	- 010
3	- 011
4	- 100
5	- 101
6	- 110
7	- 111