Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

Lab 2
C programming (II)

Content

1 Objectives

The general objective of this lab session is to know new aspects of C language and to apply them in practice, using the tools that UNIX provides to this purpose.
The concrete objectives are:

- Getting a deeper understanding of C functions and how the affect to variables use and scope.
- Practicing several language features like: command line arguments, strings, pointers and structures.

You will achieve this working on C programs, that you will complete or change following the requirements and you will analyze the results got.

1.1 Working environment

The working environment is the same used on the former session: Linux and gcc compiler, already described. You can use any available plain text editor, but we recommend using **kate**.
You can also do this lab work on Mac OSX, using gcc or XCode. There is also a gcc version for Windows but we don't recommend it as it differs from the Linux version.
You can find in PoliformaT the code templates for the exercises on folder "src english".

2 Functions in a C program

On the former session you have started creating C programs were you have noticed that, as in Java, there is a function named `main` where the program execution starts. Apart from this function you will find on C programs another two kinds of functions: those implemented inside the file and those provided by libraries, i.e. `printf()` defined on `stdio.h`. On figure 1 you can see the code of "circle.c" that contains only function `main()` you are going to compile and execute it, then on the next section you will put some code from main() into another function.

```
#include <stdio.h>
#define PI 3.1416

main() {
   float area, radius;
   radius = 10;
   area = PI * (radius * radius);
   printf("Area for circle with radius %f es %f\n", radius, area);
}
```

Figure 1: Code in file "*circle.c*"

As you did before the command to compile and to generate the executable file is:

```
$ gcc -o circle circle.c
```

Check that the executable has been generated with command `ls –l`. To run the file got from the former command you have to do:
Note. Prefix ./ is required because in Linux the working directory is not included on the PATH variable.

```
$ ./circle
```

2.1 Exercise 1: defining and declaring functions

This section shows how to use functions inside a C program. To achieve this you have to create a new file on **kate** named `circle2.c` and to copy in it the content got from `circle.c`. At the end of `circle2.c` you have to add the definition of function `areaC()` and so this computation is got apart from `main()`. This function has the circle radius as input parameter and return the computed circle area. So you have to replace in `main()` line:

```
area = PI * (radius * radius);
```

by:

```
area = areaC(radius);
```

Compile now "circle2.c" and look at the messages that appear on the compilation output:

```
$ gcc circle2.c –o circle2
```

2.2 Questions: errors or warnings

a) The messages indicate errors or warnings?
b) How do you interpret the messages?

In order to avoid the observed problems you can follow two ways. First changing the functions declaration order putting areaC() before main()as shown in figure 2. If we compile `circle2.c` with this change we have to get the executable file without errors. The second way is leaving the areaC() implementation after main() but writing before main() the areaC() function declaration with line:

```
float areaC(float radius);
```

As you can see only the output type, the name of the function and the parameters are specified. In summary, in order to avoid compilation errors, functions different from main() have to be either defined or declared before main().

```
#include <stdio.h>
#define PI 3.1416

float areaC (float radius) {
   return (PI * (radius * radius));
}

main() {
   float area, radius;
   radius = 10;
   area = areaC(radius);
   printf("Area for circle with radius %f es %f\n", radius, area);
}
```

Figure 2: Declaring function areaC() before main() in "*circle2.c*"

2.3 Variable scope

One of the aspects to notice in “circle2.c” code, shown on Figure 2, is the appearance of variable “radius” in two different locations: function areaC() and function main(). This is possible due to how variable scope works in C:

- **Global variables**: They are declared outside any function, and can be accessed from any function implemented inside the file.
- **Local variables**: They are declared inside a function and then they are only accessible inside the function. This variables are not persistent and so they lose their value once the function ends.
- **Static variables**: They are local variables but persistent, so they keep their final value for the next function call.

One of the most important rules that affects local variables applies when there are global and local variables with the same name as, an example is shown in figure 3 that contains “variables.c” code. In this situation, local variables have priority over global. In order to verify it compile “variables.c” and analyze its result. What value do you guess that will be displayed on screen for variable x?

```
#include <stdio.h>

int x=2;

void m(){
    x = 4;
}

void main(){
    int x=3;
    m();
    printf("%d", x);
}
```

Figure 3: Code in “*variables.c*”

2.4 Exercise 2: working with variables

Source code of program “variables2.c”, shown in Figure 3, intends to review situations that can arise when using variables in different scopes.

```
#include <stdio.h>

int a = 0; /* global variable */

// This function increases the value of global variable a by 1
void inc_a(void) {
    int a;
    a++;
}

// This function returns the previous value and saves the new value v
int former_value(int v) {
    int temp;
    // Declare here static variable s

    temp = s;
    s = v;
    return b;
}
main() {
```

```
    int b = 2; /* local variable */
    inc_a();
    former_value(b);
    printf("a= %d, b= %d\n", a, b);
    a++;
    b++;
    inc_a();
    b = former_value(b);
    printf("a= %d, b= %d\n", a, b);
}
```

Figure 4: Code to correct on file "*variables2.c*"

Compile `variables2.c` with:

```
$ gcc -o variables2 variables2.c
```

You will get some errors that are commonly shown following the format shown next:

```
variables2.c:18:14:error: 's' no se declaró aquí (primer uso en esta función
```

File name

Line number where the error or warning is located

Indicates if it is an error or a warning

Error or warning description

The program also contains errors to be corrected:

- Function `inc_a` has to work with global variable *a*, so it shouldn't be defined as local.
- On `former_value` you have to define variable *s* as indicated by the comment, and in order to actually return its former value the function has to return `temp`, not b.

After program execution the following output appears on the console:

```
a= 1, b= 2
a= 3, b= 2
```

2.5 Questions

a) Explain the change in the value of variable "a" and why it is increased to value 3.
b) Why variable "b" keeps its value?

3 Command line parameters

When executing a command in UNIX it is common to pass parameters. In a C program, we can treat these parameters in a very simple way with `argc` and `argv` variables. To be able to use these variables the `main` function must be defined with these two arguments like:

```
int main (int argc, char *argv [])
```

a) `argc` contains el number of arguments passed, it will always be greater than zero, as the first argument is always the command name.
b) `argv` is a vector of strings containing the arguments. The first element of this vector (argv[0]) will always be the command name.

In this exercise you have to do two programs using the following program "`arguments.c`" as starting point:

```
#include <stdio.h>

int main(int argc, char *argv[])
{


     // To be completed…


}
```

Figure 5: Initial content on file "*arguments.c*"

3.1 Exercise 3: program arguments

Implement a program "arguments.c" that will show on the screen the number of arguments and their values. You have to do a loop that uses printf() to display every provided argument "argv[i]", relying on "argc" value. Below you can see "arguments" execution result for two cases of arguments passed:

```
$ ./arguments
Number of arguments = 1
  Argument 0 is ./arguments

$ ./arguments one two three
Number of arguments = 4
  Argument 0 is ./arguments
  Argument 1 is one
  Argument 2 is two
  Argument 3 is three
```

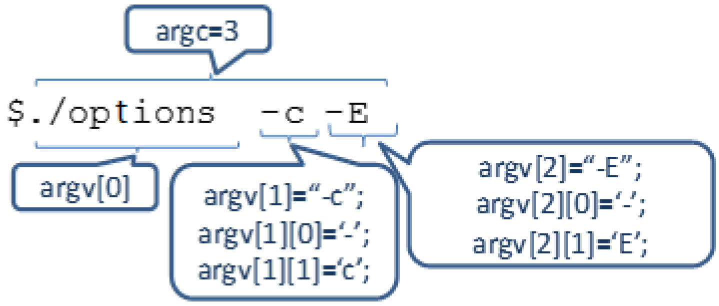
3.2 Exercise 4: program options

Implement program "`options.c`" able to identify the following options (similar to the ones on gcc). If an option has an associated path it has to be shown:

```
-c   will show "Compile"
-E   will show "Preprocess"
-i will show "Include + path"
```

Below you can see the result that you have to get after the program execution with the arguments in each case:

```
$ ./options -c
   Argument 1 is Compile
$ ./options -c -E -i/includes
   Argument 1 is Compile
   Argument 2 is Preprocess
   Argument 3 is Include /includes
```



4 Pointers and structures

A pointer is a variable that contains the address of another variable. This allows us to access and modify elements of strings and structures in an easy way. In this part of the lab session we will complete small programs that deal with pointers, strings and structures.

4.1 Exercise 5: program uppercase

Complete the following program `uppercase.c (figure 6)`, which has to convert a text read from the console into uppercase and then show the result on the screen. In particular you have to:

a) Define variables `string` and `string2` as char vectors of SIZE_STRING size.
b) Read a text from the console and assign it to `string`. To read strings that contain spaces you can use `scanf("%[^\n]s", str).`
c) Complete the conversion to uppercase loop. To achieve this make use of two pointers to strings p1 and p2, where p1 points to string and p2 points to string2. Therefore element pointed by p1 will be copied to the element pointed by 2 (subtracting 32 to convert it to uppercase, only in case of being a lowercase character). At the end of the loop a null value has to be appended to string2.
d) Print in the console string2, that will contain the text converted to uppercase.

```
#include <stdio.h>
#define SIZE_STRING 200

main() {
    // Character pointers to copy the input string
    char *p1, *p2;

    // A) Define the string variables string and string2

    // B) Read string in the console

    // C) Convert to uppercase
    p1 = string;
    p2 = string2;
    while (*p1 != '\0') {
        // Copy p1 to p2 subtracting 32 if necessary
    }
    // Remember to append the null value at the end of string2

    // D) Out in the console string2.
}
```

Figure 6: Initial "uppercase.c" content

4.2 Exercise 6: program addrows

Complete the following program `addrows.c (figure 7)` that adds a series of rows and returns the addition result for every row and the total row addition. Each row is a structure containing two members, a vector with the row data and the row addition result. Do the following completions in the provided program:

a) Define a vector "rows" of structures `ROW` with size `NUM_ROWS`
b) Implement function add_row. This function is passed a pointer to the row to add. You will have to add the vector data and to assign the addition result to `suma` structure member.
c) Complete the loop to add all rows. You should call add_row passing to it the row to add. Finally complete `printf` and update variable `total_add`.

```
#include <stdio.h>
#define SIZE_ROW 100
#define NUM_ROWS 10

struct ROW {
    float data[SIZE_ROW];
    float addition;
};
// A) define a vector “rows” of structures ROW with size NUM_ROWS

void add_row(struct ROW *pf) {
// B) Implement add_row
}

// Initilize rows with value i * j
void init_rows() {
    int i, j;
    for (i = 0; i < NUM_ROWS; i++) {
        for (j = 0; j < SIZE_ROW; j++) {
            rows[i].data[j] = (float)i*j;
        }
    }
}

main() {
    int i;
    float total_add;

    init_rows ();

    // C) Complete the loop
    total_add = 0;
    for (i = 0; i < NUM_ROWS; i++) {
        // Call add_row
        printf("Row %u addition result is %f\n", i, /* TO BE COMPLETED */);
        // update total_add with the actual row
    }

    printf("Final addition result is %f\n", total_add);
}
```

Figure 7: Initial "addrows.c" content

When executing the program the output should be:

```
$ ./addrows
Row 0 addition result is 0.000000
```

```
Row 1 addition result is 4950.000000
Row 2 addition result is 9900.000000
Row 3 addition result is 14850.000000
Row 4 addition result is 19800.000000
Row 5 addition result is 24750.000000
Row 6 addition result is 29700.000000
Row 7 es 34650.000000
Row 8 es 39600.000000
Row 9 es 44550.000000
Final addition result is 222750.000000
```

5 Parameter passing by reference

To conclude this session, we will introduce the parameter passing by reference to a function, that will allow to return one or more values from function execution. Examples of functions such as areaC() in "circles2.c" (Figure 2) or former_value() in "variables2.c" (Figure 4) use parameter passing by value. In case of areaC() the parameter is of type float (input parameter) and in case of former_value() it was an int. When parameters are passed by reference, what is passed to the function is not the actual value of the variable but its address in the form of a pointer. For example, addrow() in "addrows.c" (figure 7) has as parameter a pointer to a structure "struct ROW *pf". The main contribution of this mechanism is the possibility to modify inside the function the actual value of the variable, so the parameter can act both as input or output. The following example uses parameter passing by reference to set the value of variable "c":
Note. Notice that in the second printf() variable "c" is passed to F as "&c", so the address of "c" is passed.

```
#include <stdio.h>

char F(char *c){
        c[0] = 'f';
        return (*c);
}
main () {
        char c;
        c = 'a';
        printf("%c\n",c);
        printf("%c\n", F(&c));
        printf("%c\n", c);
}
```

Figure 8: outputparameter.*c*

5.1 Exercise 7: program addrows2

In the following exercise, you have to modify "addrows.c", described in section 4.2, into "addrows2.c" that comply with the following requirements:

a) Variable "rows" in now defined inside `main()`.
b) Change function `init_rows()` into init_row(), in such a way that a pointer to the row to be initialized is passed to init_row() and it will be called from `main()` for every row before calling to `add_row()`.

You have to get the same final result than with "`addrows.c`".