Fundamentos de los Sistemas Operativos

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

fSO

Lab 1
C programming (I)

Content

1 Objetives

Editing programs in C language and compiling them on LINUX with gcc (GNU Compiler Collection).

- Working with programs in C contained in a single file
- Learning how to compile and run simple programs
- Detecting basic compilation errors
- Learning to communicate with the program reading and writing in the terminal
- Getting experience with " if.. .else if ", "switch" and "while"

Warning: This lab session assumes that students already know programming in another language.

2 Tools

In order to create programs in C you need a plain text editor and a C compiler. The lab environment is Linux with gcc compiler. There are several text editors available we recommend to use *kate*. In Annex 1 you can look at concrete aspect of gss and the I/O functions printf() and scanf().

2.1 Steps to create a C program

¡IMPORTANT!

Be aware that the following steps to create C programs will be done on most of the FSO lab sessions:

- Write the program source code with a text editor and save it with extension *". c"*, for instance:
  ```
  $ kate myprogram.c &
  ```
- If you use the option "- o" with the compiler and there are no errors in the source code, then a file with the executable code is generated. Therefore, we will provide to the compiler at least two file names: source code file and executable file, for instance:
  ```
  $ gcc myprogram.c  -o codeprogram
  ```
- To execute the program just type the executable file name with prefix "./" and press enter:
  ```
  $ ./codeprogram
  ```

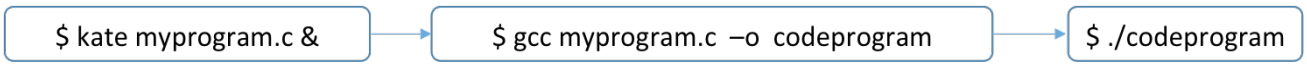

Figure 1: Steps to create a C program

¡REMEMBER! A program in C always needs a main() function. In C all variables have to be declared before using them and the instructions are written in lower case letters and every sentence ends with ";".

2.2 Warnings y errors

It is important to differentiate between warnings and errors when viewing the result of a program compilation. A warning indicates that the code is ambiguous and that it can be interpreted differently from one compiler to another, but the executable is created. An error indicates it hasn't been possible to compile or link successfully the program and therefore the executable is not created.

Typical compilation errors usually occur when something is omitted in the source code as such a ";". Linking errors are more subtle and often occur when the program is partitioned into several files (a later study) or when using libraries.

2.3 Looking for information with *"man"*

"man" (manual) is an standard UNIX tool that has an entry for every available command, function and system call. To know everything about using man look at the output of the following command:

```
$ man man
```

System calls documentation is on section 2 of the manual, while C functions are in section 3. For example, to get information about function printf() or sqrt() you have to type:

```
$ man 3 printf
$ man 3 sqrt
```

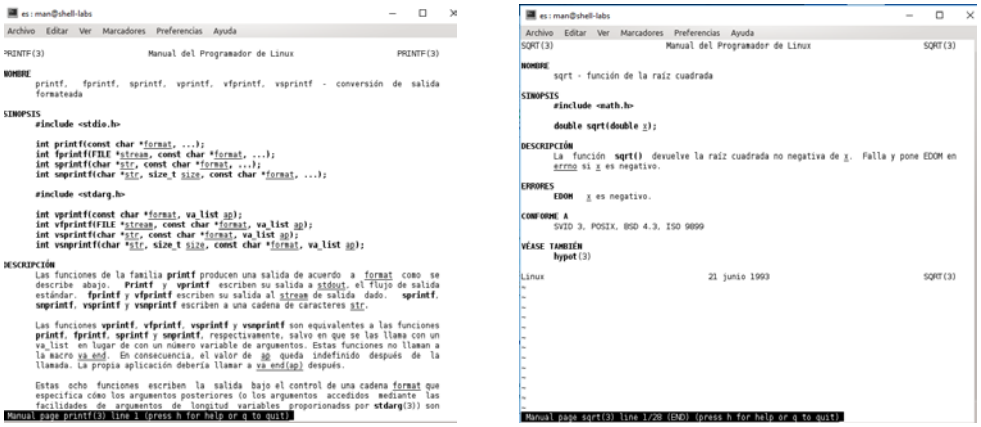

Figure 2: Man pages for C functions *printf()* y *sqrt()*.

On every manual page there is a SYNOPSIS section that includes information about using the function. For example, in case of Figure 2:

#include <stdio.h>

means that in order to use function printf() you have to add the #include line in the program file. It also describes the parameters required to invoke the function and what it is returned after its execution. The DESCRIPTION section provides a short description of what the function does. To exit a man page you have to press "q".

3 Hands on

Learning C programming means do actual C programs, try them and experiment with them.

¡IMPORTANT! Remember to keep always a copy of the version of your programs when they work and before making big changes.

3.1 Exercise 1.1 Creating a C program: Edit, compile and execute

Create file “numbers.c” with the content shown on figure 3, which corresponds to the basic structure of a C program. The "numbers" program writes numbers on the terminal, so the <stdio.h> library is included in the source code and function *printf* is called. As explained before these are the steps to follow:

Start opening the date editor with:

```
$ kate numbers.c &
```

Once kate opens write the program sentences from figure 3. Remember saving the file from time to time to avoid losing your work.

Go back to the terminal windows (press enter to recover it) and compile with:

```
$ gcc numbers.c  -o numbers
```

Check if the compiler shows errors (warnings are also worth to look at).

You can use “-v” option to get a verbose compilation description.

After compiling without errors you can execute your program with:

```
$ ./numbers
```

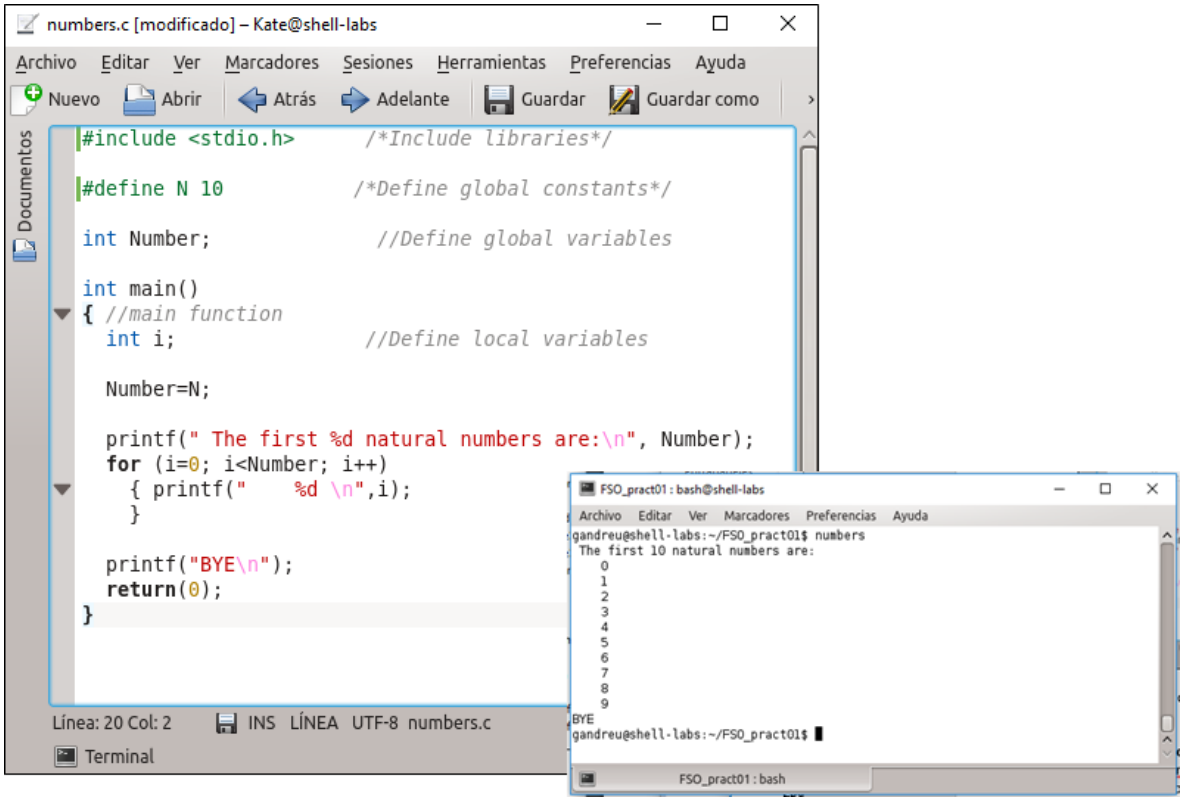

Figura 3: Content of file “numbers.c” inside the kate editor and the terminal windows with the execution result.

3.2 Exercise 1.2 Compiler errors and *warnings*

Edit file “*numbers.c*” and do every one of the changes indicated on the following table. After every change save the file, compile and identify the error with the message shown by the compiler. Then fill the corresponding table row. Be aware that before every change you have to undo the previous one.

Action	Line number	The executable has been generated?	Error o warning
Comment the declaration of variable i //int i;			
Replace “%d” by “%f”			
Remove one “;”			
Replace “main” by another name like “many”			

3.3 Exercise 1.3 Reading from the keyboard: function “scanf()”

To make "numbers" program writing a different number set you have to change the value of global constant N on the source file and then to recompile. However, the C language provides other options without having to modify the source code and compile every time.

Save "numbers.c" file as “numbers2.c” and edit file "numbers2.c" so it will ask to the user how many numbers to show and then read the value entered by the user. Use the scanf() function (example: scanf("%d", Number)) The required behavior is shown on figure 4.

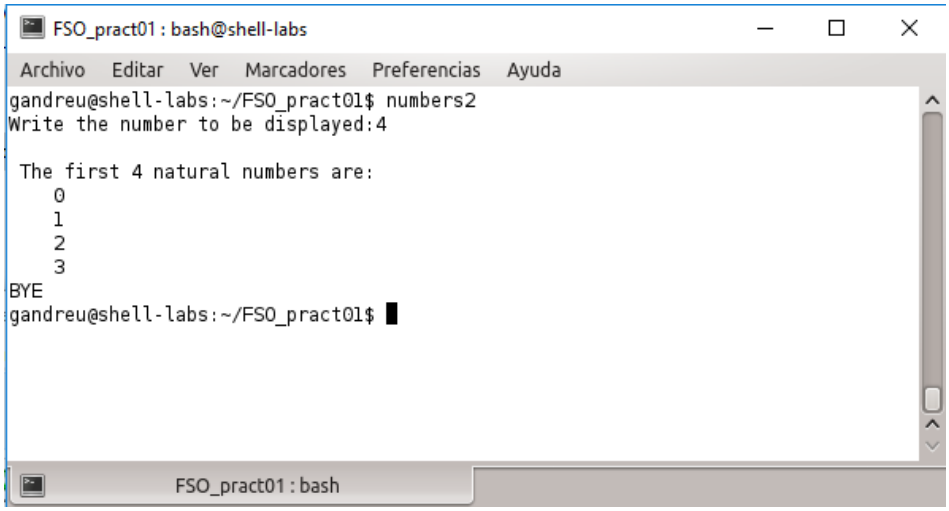

Figure 4: "*numbers2*" execution, it visualizes the amount of number set by the user.

3.4 Exercise 1.4 Execution flow control "if …. else if"

Download from Poliformat file "atm.c" (automatic teller machine), it contains the code shown in figure 5, then compile it and run it. "atm" simulates the operations of an ATM and it is written in a basic way using "if... else if...". Check its operation by running it as many times as the options it has.

```
#include <stdio.h>

#define InitBalance 1000
float Balance;

int main()
{ int operation, value;
  float income, withdraw;

  printf("\nWelcome to the FSO ATM\n");
  Balance=InitBalance;
  operation=0;
  printf("\nIndicate operation to do:\n");
  printf(" 1.Cash Income\n 2.Cash Withdrawal\n 3.Balance Enquiry\n");
  printf(" 4.Account Activity\n 5.Change PIN\n 6.Exit\n\n");
  printf(" Operation:");
  value=scanf("%d",&operation);

  if(operation==1){
    printf(" Cash Income\n");
    printf("\n Enter the amount to deposit:");
     scanf("%f",&income);
     Balance=Balance+income;
    printf(" Successful income\n");

  } else if(operation==2){
    printf(" Cash Withdrawal\n");
    printf("\n Enter the amount to withdraw:");
    scanf("%f",&income);

     if(Balance>income){
       Balance=Balance-income;
     }else{
       printf(" Operation does not allowed\n");
       printf("   Not enough cash\n");
     }

  }else if(operation==3){
   printf(" Balance Enquiry\n");

  }else if((operation==4)|| (operation==5)){
   printf(" This operation is not implemented");

 }else if(operation==6){
   printf(" EXIT\n");

  }else if(operation>6){
   printf(" ERROR: This opertaion does not applied\n");

  }

  printf("\n\n Current Balance: %.2f Euros", Balance);
  printf("\n\n Thanks \n\n");
   return(0);
}
```

```
gandreu@shell-labs:~/FSO_pract01$ atm

Welcome to the FSO ATM

Indicate operation to do:
 1.Cash Income
 2.Cash Withdrawal
 3.Balance Enquiry
 4.Account Activity
 5.Change PIN
 6.Exit

 Operation:2
 Cash Withdrawal

 Enter the amount to withdraw:270


 Current Balance: 730.00 Euros

 Thanks

gandreu@shell-labs:~/FSO_pract01$
```

Figure 5: Source code of "atm.c" and execution example.

3.5 Exercise 1.5 Execution flow control "switch"

For multiple choice controlled by matching a variable on a set of values, it is good to use the following structure:

```
switch(exp) {
  case exp-const: ------------------------;
       break;
 case exp-const: ------------------------;
       break;
 default: -----------------------------------;
       break;
}
```

Copy file "atm.c" into "atm2.c" using the shell command:

```
$ cp atm.c atm2.c
```

Modify "atm2.c" using the *switch{}* structure replacing the "if … else if" sentences that check the value of variable "oper" by the corresponding "case" statement. The operation has to be the same as with "if... else". Once you've done it right answer the following questions:

Question	Answer
What expression (exp) have you used on *switch*(exp)?	
Why do you need to do *break*?	
How have you solved the checking on line 43 of the original code? **Note.** Activate line number in kate: Ver -> Mostrar números de línea	
Why is needed the *default* entry?	

3.6 Exercise 1.6 Loop "while"

Copy "*atm2.c*" to "*atm3.c"*. Modify atm3.c using the structure *while (exp) {}*, so it will allow more than one operation on every execution, until the exit option (6) is selected. Once you'll have it working try to answer the following questions:

Note. To indent multiple lines at the same time select the lines to indent and press on Herramientas -> Alinear

Question	Answer
What expression (exp) have you used on while(exp)?	
Have you needed new sentences? Why?	

4 Homework suggestions (optional)

The program "atm.c" can be improved, you find next some functions you can try to implement:

- Balance: Add the option whether to show or not the balance (rely on functions from *string.h* library).
- Language: Add the option to select the menu language.
- Safety: Add user authentication.
- Show account movements: The user has to be able to see his/her former movements, in this case you have to work with files so the account will be persistent and so the log file.

5 Annex 1: GCC ("GNU Compiler Collection")

GCC ("GNU Compiler Collection") includes a set of compilers developed within the GNU project, therefore, it is free software. It currently includes compilers for C, C++, Objective C, Fortran and Ada. GCC takes a source code file and generates a binary executable program for the running platform. GCC can generate code for several CPUs like Intel x 86, ARM, Alpha, Power PC, etc. It is used as development compiler on most platforms. For instance Linux, Mac OS X, iOS (iPhone and iPad) are entirely compiled with GCC.

5.1 *gcc* syntax and opctions

The gcc use syntax is as follows:

```
$ gcc [-options] [source_files] [object_files] -o output_file...
```

Options are preceded by a dash, as it is usual on UNIX, each option may consist of several letters and several options can't be grouped on a single dash. Some options require a file name or directory, others don't. There may be multiple file names to be included in the compilation process.
GCC has a lot of options, next you can find some of them:

Option	Description
-c	Preprocessing, compilation, assembly producing a binary object file (.o)
-S	Preprocessing and compilation producing an assembly language file
-E	Preprocessing only sending the output to the standard output (terminal)
-o file	Specifies the output file name
-Ipath	Indicates the directory path for the header files included on the source file. No spaces between -I and path, for instance: -I/usr/include. By default there is a set of standard header paths already set
-Lpath	Indicates the directory path for the library files used on the source file. No spaces between L and path, for instance: -L/usr/lib. The standard library path don't need to be specified
-lNAME	NAME: library to link with the program. Tells the compiler what libraries to link with the working program, expanding the set of default libraries used by the linker. For instance: Option -lm includes the math library lib**m**.so
-v	Sets verbose on, so gcc shows a complete description of every compilation step

5.2 Compilation phases

On gcc, as well as other compilers, it is possible to distinguish 4 stages in the build process (Figure 6):

- **Preprocessing**: This stage interprets preprocessor directives. Among other things, #define declarations are replaced in the code by their values and it includes the source from #include declarations.
  Example;

```
$ gcc -E numbers.c > numbers.pp
 Look at numbers.pp with:
$ more numbers.pp
 You can see that variable N has been replaced by its value
```

- **Compilation**: It transforms the C code into the processor assembly language. If the target machine is different from the development is called cross compilation.

```
   $ gcc -S numbers.c
it does the first two stages by creating the file numbers.s;  try to exam it with
  $ more numbers.s
you can notice that the result is written in assembly language.
```

- **Assembly**: It transforms the program written in assembly language into a binary executable file. It is not common to perform only the assembly in isolation; usually you do all those steps until you get the object code as well:

```
 $ gcc -c numbers.c
donde se crea el archivo numbers.o a partir de numbers.c. Puede verificarse el tipo de archivo usando el commando
where numbers.o is created from numbers.c file. You can check “numbers.o” file type with command
 $ file numbers.o
```

- **Linking**: C functions used in our code, such as printf(), are already compiled and assembled inside installed libraries. It is necessary to add in some way the binary code of these functions in our executable. This is the linking stage, where one or more object code modules gather with the library code. The linker in Linux is called *ld*. This is a example of using *ld*:

```
 $ ld -o circle circle.o –lc
```

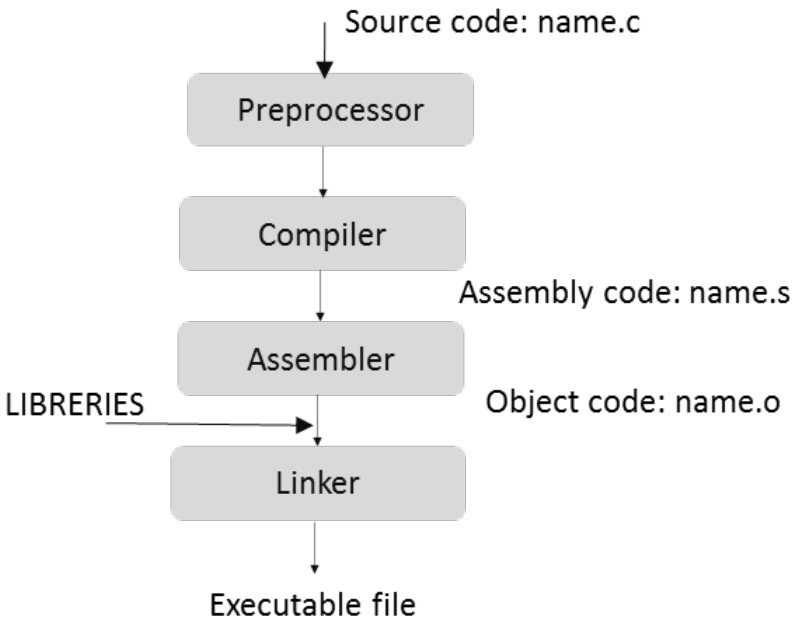

Figure 6: Compilation phases.

5.3 All phases in one single step

If your program is on a single source file the all previously described phases can be done with one single command:

```
$ gcc -o numbers numbers.c
```

In case of no errors the the executable "numbers" is directly generated.
It is interesting trying option "–v" and look at all the details given by gcc related to all the compilation phases:

```
$ gcc -v -o numbers numbers.c
```

gcc use examples:

```
$ gcc hello.c -o hello
  Compiles program hello.c and generates executable file hello
$ gcc hola.c
  Compiles program hello.c and generates executable a.out (default name)
$ gcc -o hello hello.c
  The same as the first example
```

6 Annex 2: stdio.h functions printf() and scanf()

In C input and output is done using the standard functions provided by the C standard library. In order to use them you have to include the header "stdio.h". Two function very often used are "printf()" and "scanf()", many others are available.

6.1 printf()

printf() sends a formatted string to the terminal (default standard output). The first printf() parameter is a formatted string (control string) followed by a sequence of arguments. The formatted string contains two types of objects: ordinary characters and conversion specifications. Ordinary characters are copied as they are to the output stream, while conversion specifications cause the conversion and printing of the argument values. Each conversion specification must be preceded by character %.

The calling syntax is:

```
printf(<control string> [, <arguments list> ] )
```

Example:

Source code	Output on the terminal

`#include <stdio.h>`
`int main(){`
`  int num1 = 10;`
`  printf("Number %d", num1);`
`  return 0;`
`}`
	`Number 10`

Figure 7: Example of a program with printf()

The <control_string > is a sequence of characters that has to be written between double quotes (""). When a program calls printf() with an <arguments_list> every argument value (if there is no error) is shown on the terminal along with the <control_string>. Therefore, the <control_string> indicates the output data format to be displayed and it can consist of:

- Ordinary text

- Formatting specifications
- Scape sequences

Table 1: printf() formatting specs

	Argument type : output
%c	**char** : writes a single character
%d, %i	**int** : writes an integer in decimal
%e, %E	**float, double** : writes a floating point number in scientific notation indicating the exponent with prefix “e”
%f, %F	**float, double** : writes a floating point number in fixed point notation “xxxx.ddd”
%g, %G	**float, double** : writes the shortest option between "%e" and "%f" or between "%E" and "%F"
%o	**int** : writes an unsiged integer in octal
%s	**char *** : writes a character array until finding character ‘\0’
%u	**int** : writes an unsigned integer in decimal
%x, %X	**int** : writes an unsigned integer in hexadecimal (without 0x prefix)
%p	**void *** : writes the value of a pointer (that always is a memory address)

Format specifiers provide the screen output format for the arguments. Escape characters allow us to control how the output is displayed. They consists of the backslash character (‘\’) followed by another character. Escape character can be of one of the following two types:

- Graphic: they are shown as characters on the screen
- Non graphic: they provide action control on the output

Table 1: Escape characters most often used.

Escape characters	**Meaning / Action**
\n	New line (ASCII 010)
\t	Horizontal tab (ASCII 009)
\v	Vertial tab
\f	New page
\r	Carriage return (ASCII 013)
\\	Shows character \ (ASCII 092)
\”	Shows character ” (ASCII 034)

6.2 scanf()

scanf() allows assigning values to variables from the keyboard, its syntax is similar to printf(). It can assign values to one or more variables reading one or more values from the standard input (the keyboard by default). The scanf() calling syntax is:

scanf(<control_string>, <arguments_list >)

The <control_string > specifies the data entry format that will be picked up from the keyboard. This is done again with format specifiers that are the same as the ones on printf(). For intance:

scanf("%c %d %f %s", &ch, &i, &x, cad);

Source code	**Output on the terminal**

`#include <stdio.h>`
`int main(){`
`  int num;`
`  printf("\n Write an integer: ”);`
`  scanf("%d”, &num);`
`  return 0;`
	Write an integer: 4

`}`	

Figure 8: Program example with scanf().

When running the program in figure 8 variable *num* has memory assigned. And if, when prompted to type an integer, the user types for instance value 4, then the output will be the one shown in the right column of figure 8. Since the variable type of *num* is integer, then the format specifier in the control string is %d. Notice that parameter *num* is preceded by operator &, that is *num* is passed by reference because it is an output parameter.