Fundamentos de los Sistemas Operativos

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

fSO

Lab session 2
Compiling C programs

Content

1 Goals

The aim of this lab session is to learn C programming using UNIX tools. Specifically:

- To start working with the UNIX C programming environment (shell, editor, gcc, make, etc.).
- To know the compiling phases of a C program.
- To program in C, using the characteristics that differ from Java (variable scope, strings, pointers, memory management, etc.) .

To achieve this the student will make and/or modify simple C programs.

1.1 Working environmet

The environment required by this practice is a Linux PC with gcc (GNU Compiler) installed. The editor can be KEdit, KWrite or any other one available. This practice also can be done on Mac OS X using gcc and some of the available editors (XCode recommended)[1].

On the FSO site in Poliformat you can find a tar file "lab02_src.tar" you will find a set of C source files to deal with along this lab session.

To edit the files you can use one of the several editors available. We recommend kate that can be started from the program menu or from the terminal with the command:

```
$ kate Circle.c &
```

2 Introduction to GCC

2.1 GCC compiler

GCC is a set of compileris developed by the GNU Project and therefore is all of them are free software. In the beginning there was only a C compiler named GNU C Compiler. Nowadays there are compilers for several languages like C, C++, Objective C, Fortran, Ada and Java. Therefore, GCC now stands for "GNU Compiler Collection". GCC is capable of receiving a source program, in any of the fore mentioned languages, and generating a binary executable program in native machine language where it is intended to run. Therefore GCC can generate code for different architectures including Intel x86, ARM, Alpha, Power PC and many more. This is why it is used for native application development on most platforms. For instance, Linux, Mac OS X, IOS (iPhone and iPath) operating systems are compiled with GCC.

2.2 GCC syntax and options

The syntax for using the gcc compiler is:

```
$  gcc [ option | file ] ...
```

The options are preceded by a hyphen, as is usual in UNIX. Every option can have several letters and several options cannot be grouped in the same hyphen. Some options require a file name or directory, others don't. Finally, there may be multiple file names included in the compilation process. As an example see the following command, It generates the compiled **hello** program from the source program **hello.c**

```
$  gcc -o hello hello.c
```

[1] NOTA: There is a gcc version for Windows but it differs from the one described here, then its use is not recommended.

Table-1. GCC has lots of options. Next we describe in detail the ones use along FSO lab sessions.

Opción	Descripción
`-c`	Does preprocessing, compilation and assembly, to give the object code file with extension .o
`-S`	Does preprocessing and compilation, to give the assembly file
`-E`	Does only preprocessing, then sends the result to the standard output
`-o file`	Specifies the output (executable) file name.
`-Ipath`	Specifies a path to the directory to look for files marked to be included in the source file. There is no space between I and the pathj, like: `-I/usr/include`. By default standard include paths need not be specified
`-Lpath`	Specifies a path to the directory to look for objec code files that include functions referenced by the program. There is no space between L y and the path, like: `-L/usr/lib`. By default standard library paths need not be specified
`-lname`	**name** is a library to be linked with our code. This option tells the compiler what libraries to be included with our developed code. It is used when we want to increase the library set that the linker uses by default. i.e. "-lm" will include librery lib**m**.so (prefix **lib** and extensión **.so** are excluded)
`-v`	Verbose on; shows the commands executed in every compilation step indicating their versions

2.3 Compilation phases

GCC, like other compilers, has four phases in the compilation process:

- **Preprocessing**: Preprocessor directives interpretation. Among other things, initialized variables with #define are replaced in the code by their values in all places where they appear. Also the source of #include sentences is included.
- **Compilation**: transforms the C code into the assembly language corresponding to the processor in your machine. Normally the target machine is the same as the one used for programming, but if this is not the case, then the process is called cross-compilation.
- **Assembly**: transforms the program written in assembly language to object code, a binary machine language file executable by the processor.
- **Linking**: The C / C++ functions included in the code (i.e. printf()), are already compiled and assembled into libraries available in the system. The binary code of these functions should be incorporated in some way into our executable. This constitutes the linking step, where one or more object code library modules are bind with the program code.

3 Exercise 1: Compilation steps and modular structure

3.1 Compilation phases

In this exercise we will try and compile a C program and we will see the several compilation steps. The program to compile is "Circle.c". It is very simple and it is shown below:

```
#include <stdio.h>

#define PI 3.1416
main() {
   float per, radius;
   radius = 10;
   per = 2* PI * radius);
   printf("Perimeter circle (radius= %f)= %f\n", radius, per);
}
```

Figure-1: Code inside file "C*ircle.c*"

To compile this program and to generate the executable directly, performing all compilation steps, enter the following command:

```
$  gcc -o Circle Circle.c
```

Verify that you have generated an executable file (with ls -la). To test the program simply type the following command:
Note. You should put "./" before the filename in order to allow the shell finding the executable in the current directory instead of using PATH.

```
$  ./Circle
```

This will be the common way to generate an executable file from a source code file. But let's see the results for every compilation phase.

a) PREPROCESSING: We can preprocess the file with the –E option. You have to redirect output to a file, that we will name " circle.pp".

```
$  gcc -E Circle.c > Circle.pp
```

Have a look to the file "circle.pp" (using kate) and we can verify two things: **a)** #include has disappeared and its has been replaced by the included file content with a series of definitions and function prototypes for input and output (printf function) and **b)** at end of file you can check that the variable PI has been replaced by its value, 3.1416, as it was fixed by #define.

b) COMPILATION: We can compile (and preprocess) the source file with –S option that will generate assembly code for the compiled program.

```
$  gcc -S Circle.c
```

We can see that a file with .s extension has been created that contains the assembly code. Open this file and see that it is indeed the case.

c) ASSEMBLY: the assembly process transforms the assembly code into a binary machine code that is executable by the processor:

```
$  gcc -c Circle.c
```

You can check the file with the command:

```
$  file Circle.o
```

d) LINKING: our program uses **printf** that is defined in the **stdio** library. To include this function into the executable, the linking step should be done. This constitutes the binding step; one or more object code modules are bind with existing code libraries.

```
$  gcc –o Circle Circle.o
```

If we don't use option "-o" the the executable file generated is named by default "a.out".

3.2 Modular structure

In this section we learn to structure a program in modules. To do this we will create a function named "area" in a separate file and we will use it in our program Circle.c
Step to follow:

a) Create a new file named `area.c` and define function "`float area(float radio)`" relying on the code shown on figure-1. Remember that this file will contain the function implementation.
b) Create a new file named `area.h` and include in it the function declaration. You only have to write the function definition or prototype.
c) Copy file `Circle.c` into `Circle2.c`
d) Edit `Circle2.c` to include file `area.h` and to change function `main` to call function "`area`" implemented on file `area.c`.
   NOTE: Notice that `area.h` is located on the working directory so you to include it using the following syntax:

```
#include "area.h"
```

To compile we will do it as follows including both source files:

```
$   gcc –o Circle2 Circle2.c area.c
```

This command performs all the compilation phases: it compiles `Circle2.c` and `area.c` generating their object files (.o) and then it will link function "`area`" to generate the executable file `Circle2`.

3.3 Questions

a) Enumerate and describe the file types generated in any one of the compilation phases.
b) In the last compilation both source files `Circle2.c` and `area.c`, are passed as arguments to the gcc compiler. Check what happens if `area.c` is not included as an argument.
c) What differences and similarities there are between compiling a C and a Java program?

4 Exercise 2: Variable scope and error handling

The goal of this exercise is twofold:

- To understand the scope of variables
- To learn how to correct errors occurred during a program compilation.

To do this we will work on a program that contains a number of errors (syntactical and logical) that the student must correct until the program works correctly.

4.1 Variable scope

In C the variable scope can be:

- Global: declared outside any function, they can be accessed from any function of the file.
- Local: declared within a function, they are only accessible within the function where declared.
- Static: the same as local but they keep their values among function calls.

The following program `variables.c` has some error to correct:

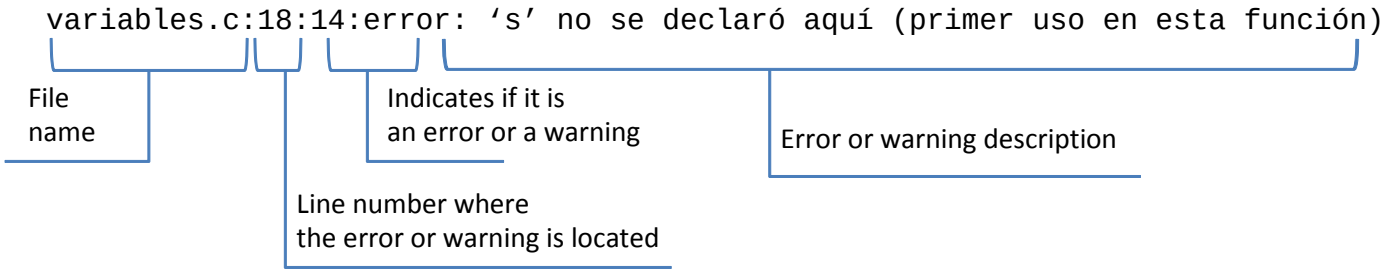
```
#include <stdio.h>

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-2: Code delivered on file "*variables.c*" to be corrected

You have to compile this program with command:

```
$ gcc -o variables variables.c
```

Some errors will appear. Errors use to have the following format:



A distinction must be done between warnings and errors:

- **Warnings** are messages that the compiler provides because it 'thinks' that there are some mistakes or standard mismatch. If there are only warning messages when compiling then the executable is generated, although it is advisable to check the code to see if it is possible remove the warnings.
- In contras **errors** must be fixed in order to get the executable file.

The program also contains a number of programming errors to be corrected. After program execution the following output appears in the console:

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

4.2 Question

a) What changes would you make in the previous program for the global variable “a” to be declared local to the `main()` function?

5 Exercise 3: Command line parameters

When executing a command in UNIX it is common to pass it 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[]) {

     // main function code

}
```

Figure-3: File content of “*arguments.c*”

5.1 Listing the command arguments

Implement a program (`listArgs`) that will show on the screen the number of arguments and the content for one. Below is what should be the result of the program execution with different arguments:

```
$ ./listArgs
Argument number = 1
  Argument 0 is ./listArgs
$ ./listArgs une two three
Numero de argumentos = 4
  Argument 0 is ./listArgs
  Argument 1 is one
  Argument 2 is two
  Argument 3 is three
```

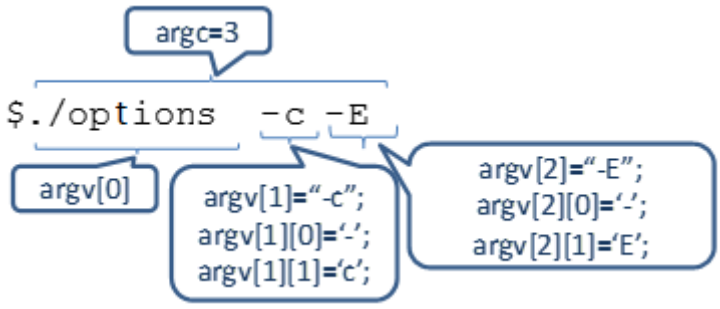
5.2 Dealing with options

Implement a program (`options.c`) able to identify the following options, similar to the gcc ones, and if so it will show the associates option path:

```
-c   will show “Compile”
-E   will show “Preprocess”
-iruta will show “Include + path”
```

Below is what should be the result of execution with different arguments:

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



6 Exercise 4: Pointers and structures

A pointer is a variable that contains the address of another object. 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.

6.1 Program uppercase

Complete the following program `uppercase.c` (`figure 4`), 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`.
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.
}
```

Figura-4: Initial code for “uppercase.c”

6.2 Program addrows

Complete the following program `addrows.c` (`figure 5`) 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 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 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;

    inicia_filas();

    // 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);
}
```

Figura-5: Initial code for “addrows.c”

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
```