Systems Programming

Laboratory 1 – C revision

Learning objectives

Arrays / Pointers

Pointers to funcctions

Dynamic loading of libraries

In this laboratory students will do a revision of C programming and learn how to dynamically load and call functions in C:

- Arrays
- Pointers
- Compilation of programs with multiple files
- Dynamic loading of libraries
- Pointers to functions

Table of Contents

1 Array of strings (argv).....2

2 Exercise 1.....2

3 Exercise 2.....2

4 Compilation of multiple files.....3

5 Exercise 3.....3

6 Exercise 4.....3

7 Pointers to functions.....3

8 Dynamic libraries.....4

9 Exercise 4.....5

9.1 Compilation of the external libraries.....5

9.2 Loading of the external libraries.....5

1 Array of strings (argv)

When implementing C programs for Unix/Linux and other desktop operating systems it is possible to send arguments from the command line:

> copy *.c d:

or

> gcc teste.o

the name of the program being executed and the supplied arguments are given to the C code as two parameters of the main function:

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

or

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

The operating system puts he user supplied arguments in the **`argc`** and **`argv`** parameters:

- argv is a vector of strings. The first string in the name of the program
- argc in the number of elements of argv

2 Exercise 1

Implement a program that concatenates all its arguments supplied by the user in the command line into a single string using functions from the **`string.h.`**

The result of this program should be stored in a single array of characters (**`result_str`**). After the construction if this array, it should be printed in the screen with a single **`printf`** instruction.

3 Exercise 2

Rewrite the previous program without using any function from the **`string.h`** library.

4 Compilation of multiple files

https://www.cs.swarthmore.edu/~newhall/unixhelp/compilecycle.html

When a programmer issues the **gcc** command, if possible the compiler creates on executable. In this case the compiler translates the C code into assembly and link the resulting assembly code with all the necessary libraries.

It is possible to separate these two steps (translation to assembly and linking) by issue two different gcc command:

```
gcc -c file.c
gcc file.o – o file
```

When compile code to generate a executable it is also necessary to guarantee that the files contain the main function.

5 Exercise 3

Look at the files **lib1.c** **lib.h** p**rog1.c** from the exercise_3 folder.

- Try to compile the file **lib1.c** issuing the command **gcc lib1.c**
- Try to compile the file **prog1.c** issuing the command **gcc prog1.c**

What happened?

- How to just compile lib1.c?
- How to create a program?
- Compile the file prog1.c (and create a program) to use the lib1.c functions.

6 Exercise 4

Using the Visual Studio Code, open the previous exercise directory and try to compile both files into a executable.

VSCode will only try to compile the current file.

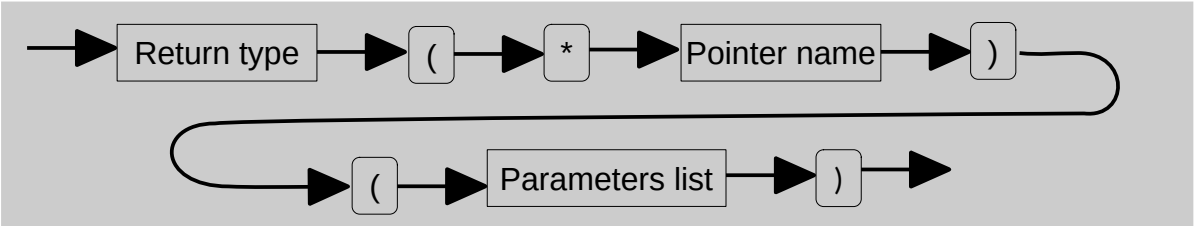
Create a Makefile so that it is possible to compile all files into a program and debug it using VSCode

7 Pointers to functions

http://beej.us/guide/bgc/html/multi/morestuff.html#ptfunc

It is possible to declare variables and function parameters that point into functions. After suitable assignment these variables can be called as regular functions.

The syntax of a declaration a pointer to function is the following:



The programmer can declare a pointer to function where he would declare any variable or argument:

```
int (*function_pointer)(int a, int b)
```

This declaration is compatible with functions like:

```
int regular_function(int a, int b)
```

To store the a any function in the variable, the programmer should do an assignment:

```
function_pointer = regular function
```

it is now possible to call the function trough a variable:

```
function_pointer(12, 14).
```

All compiler verification related to the types of the arguments are done as if it was regular function.

It is also possible to declare a type that corresponds to a function pointer:

```
typedef int (*type_pf)(int a, int b);
type ptr_f;
ptr_f = regular_function;
```

or even declare arrays of functions:

```
int (*array_ptr[2])(int a, int b)
array_ptr[0] = array_ptr[1] = callme;
array_ptr[0](12, 13)
```

8 Dynamic libraries

http://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html

http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html

Sometime it is necessary to load libraries during the execution of the program (as opposed when compiling) or call specific functions depending by their name on a string.

In this class of programs, the library code should be compiled separately and not at the same type as the main.

The main program will use pointer to functions and assign those variables to the suitable versions of the functions.

Observe the **`main.c`** program from the exercise_4 folder. Depending on the option given by the user the program will load one of the libraries and call all functions inside. The names of the function are the same on both libraries.

Try to compile the 3 files at one to see the errors.

9 Exercise 4

In order to correctly implement the program follow the next steps

9.1 Compilation of the external libraries

In order to create two dynamic libraries:

- gcc lib1.c -o lib1.so -ldl -shared -fPIC
  - creates lib1.so
- gcc lib2.c -o lib2.so -ldl -shared -fPIC
  - creates lib2.so

9.2 Loading of the external libraries

The program should first load one of the libraries depending on the user input.

These new libraries (and the internal functions) can be loaded using the **dynamic link interface library composed** of the next function:

```
man dlopen
man dlsym
```

Using the previous functions modify the **`main.c`** so that one of the libraries is loaded after input from the user and both functions (**`func_1`** and **`func_f2`**) are called