Master ROC – Bash & C
Introduction to C Programming

Sami Taktak
sami.taktak@cnam.fr

September 2023

During the laboratory sessions we will use the GNU/Linux operating system.
Start by login under OpenSuse GNU/Linux.

Part 1 : Introduction

a) Bonjour le Monde

Lets start with the common example "Hello World !" which display the string "Bonjour le monde" on its standard output.

```c
#include <stdio.h>

int main() {

  /* Print the string "Bonjour le monde" on the standard
     output followed by a new line */
  printf("Bonjour le monde\n");

  return 0; // return value of the program
}
```

Listing 1 – C Program « Bonjour le Monde »

1. Open a terminal and check that your are actually working in your home folder
2. Create a dedicated directory named `LabC` for this lab session and make it your working directory
3. Open a text editor and write the program from listing 1. Save it in a file named `hello.c` inside the previously created directory. File name of a C file must end with the extension `".c"`

To be able to execute that program, we need to compile it. To compile a C program we need to use a compiler. The generic name of a C compiler under UNIX operating systems is `cc`. Under GNU/Linux, the C compiler is usually provided by the GCC, the *GNU Compiler Collection*.

4. Run the command `cc hello.c`; The program should compile without error
5. List the content of the directory `LabC`; A file named `a.out` should have been created
6. Run the program with the command `./a.out`

b) Structure of the program `hello`

Lets look in more detail to the structure of the program in `bonjour.c`.
This program `bonjour.c` has two main parts :

- A preamble that contains here a unique line **#include**
- The definition of the fonction main

Remarque (Declaration vs. Definition)**.**

- *The* definition *of a function contains :*
  - *the function name*
  - *the name and the type of each of its arguments enclosed within parenthesis*
  - *the type of the return value*
  - *the body of the function (local variable declarations ; list of instruction to be executed)*
- *A function* declaration *contains only the* prototype *of the function which consists of the name, the type of each parameters of the function enclosed in parenthesis*[1] *and the type of the return value of the function. This allow the compiler to check that the call to a function is correctly made. For example, the prototype of the main function of the program* `hello.c` *would be :*

```
        // Declaration of the main function
        int main();
```

- **Function main()**

The main function is mandatory within a C program. It is the entry point of the program, the first function to be executed. We can that in this simple example, the main function does not take any argument and return the integer value 0.

- **Comments**

The main function start with 2 comment lines (lines 5 and 6).
In C, comments can be specified :

- either as a block starting with /* and ending with */; Using this syntax, a commend might be spread over many lines
- or as a single line comment starting with // and ending at the end of the line like on line 9

- **The printf() function**

After the 2 comment lines, we can a call to the function printf(). The printf() takes at least one parameter with consist of the *format string*, which specify what and how to print it. printf() return the number of characters than have been written. That value is ignored in the program `hello.c`.

The format string allow to describe what needs to be printed and could consist of a single constant string like in printf("Bonjour␣le␣monde\n").

7. Remove the characters '\' and 'n' from the string "Bonjour␣le␣monde\n". Recompile the program and run it again. What do you notice ?

The sequence of characters `"\n"` is an *escape sequence*. It allow to specify special characters : here the new line character `"\n"`. The table 1 present the most common escape sequences :

1. the prototype of a function might also specify argument names

Escape Sequence	Corresponding Character
`\a`	system bip (alert)
`\n`	new line ; line break
`\t`	tabulation
`\\`	the anti-slash (\) character itself
`\'`	simple quote (') character itself
`\"`	double quote (") character itself

TABLE 1 – Usual Escape Sequences

- **The instruction return**

The instruction **return** allow to specify the return value of a function and ends the function execution by returning to the calling function. Here, main() returns the value 0 which, by convention specify that the program has ran without error.

- **The directive #include**

The directive **#include** allow to include the declaration of the function printf(). Without it, the compiler wont be able to know the existence of such a function and would be unable to check if the function call is done correctly.

The printf() function is part of the *Standard C Library* and is always available with the C compiler.

Remarque. *We can find the include file needed to call a function from the standard C library by checking the corresponding manual page inside the section 3 of the system manual pages :*

```
$ man 3 printf

PRINTF(3)    Linux Programmer's Manual           PRINTF(3)

NAME
   printf, fprintf, sprintf - formatted output conversion

SYNOPSIS
   #include <stdio.h>

   int printf(const char *format, ...);

⋮
```

Part 2 : Playing with Characters

a) Reading and Writing Characters in C

```c
#include<stdio.h>

int main() {
  char c;

```

```c
6     printf("Enter a character: ");
7     c = getchar();
8
9     printf("Here is the character: %c\n", c);
10
11    return 0;
12 }
```

Listing 2 – Reading and Writing Characters in C

1. Write the program given in listing 2 into a file
2. Compile and execute that program

The program from listing 2 read a character from its standard input and writes it back on its standard output.

We can see that it calls 2 functions from the C standard library :

— the function printf () seen above

— the function getchar() which reads a character from the standard input and return its value. getchar is declared in the header file `unistd.h`).

The function getchar is called on line 8 and the read value is stored in the variable c.

The variable c is declared on line 5 **char** c;. It has the type **char**. The type **char** is generally used to store character values. In table 2, we have the common type in C ;

Remarque. *The variable c is declared in the body of the function main() and can only been accessed from within the main() function body (the block enclosed within '{' and '}'). c is a* local variable *of the function main()).*

name	Meaning
`char`	encoded on 1 byte, allow to store any characters of a 8 bits encoded set
`int`	allow to store integer values ; it size depend of the processor architecture (4 bytes on 32 bits architectures, 8 on a 64 bits)
`float`	single precision floats
`double`	double precision floats

TABLE 2 – Basic type in C

On line 10, the function printf () print the string "Here is the character : " followed by the character and a new line character.

We can see that the position of the read character is specified by a conversion specifier : "%c". "%c" will be replaced by the character encoded by the value specified in c. A format string can contain many conversion specifier, and should be followed by as many argument as there are conversion specifier in the format string.

In the following example, there are 4 conversion specifier and 4 arguments after the format string :

```c
printf("Letters go from %c to %c, and digits go from %c to %c\n", 'a', 'z', '0', '9');
```

3. Write that instruction into a C program ; compile that program and run it
4. Replace the conversion string "%c" by "%d". Recompile and execute again that program. What do you observe ?

The conversion string "%d" specified that the corresponding parameter must be interpreted an integer value. The most common conversion strings are presented in table 3.

String	Meaning
`%d`	print a integer in base 10
`%4d`	print a integer in base 10 on 4 digits
`%f`	print a float
`%4.2f`	print a float on 4 digits with 2 decimal
`%c`	print a character
`%s`	print a string

Table 3 – Conversion strings

b) Character Encoding

A character encoding associate a set of printable characters (digits, letters, graphical symbols, ...) with a set of numerical values. One of the most used encoding is the ASCII (*American Standard Code for Information Interchange*) encoding presented on figure 1.

5. Modify the previous program to display a character together with its encoded value

The function is_digit() allow to test if a character is a digit.

```
1 int is_digit(char c) {
2   if ( c >= '0' && c <= '9') {
3     return 1;
4   } else {
5     return 0;
6   }
7 }
```

Listing 3 – Function returning if a character is a digit

6. Modify the previous program so it could tells if a character is a digit or not by using the function is_digit(). Recompile the program and run it
7. On line 2 of the function is_digit, digits 0 and 9 are given enclosed within single quotes, why ?
8. Remove the single quotes, recompile and run the program again. Explain what happen. Fix the program
9. Write a function that check if a character is a letter (uppercase or lowercase). Modify the program to check it a character is a digit or a letter

c) Character Array

Array can be declared in C by specifying within square brackets the number of items a variable should contain.

```
  type name[size]
```

Each cell of a array can be accessed by specifying the index of the cell we want to have access to. index values start at 0 up to the size of the array -1.

For example, to create a array of 10 integers :

```
  int tab_entier[10]; // declaration of an array of 10 integers
  int a, b;           // declaration of 2 integer a and b

```

```c
  a = 5;                 // affectation of the value 5 to a
  tab_entier[1] = a;  /* affectation of the value of a to the cell
                         of index 1 of the array tab_entier */
  b = tab_entier[1];  /* affectation of the value of the cell 1
                         of tab_entier to b */
```

The following program read a sequence of characters, store them in an array, and finally print the content of the array on its standard output :

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define SIZE 5
5
6  int main() {
7    char tab[SIZE];
8    int i = 0;          /* index allowing us to count the number
9                            of read character initalized to 0 */
10
11   printf("Give %d characters followed by Enter: ", SIZE);
12   while ( i < SIZE ) {   /* loop to read the character (number of
13                              character to read are given by SIZE) */
14     tab[i] = getchar();
15     i++;                          // incremente the value of i by 1
16   }
17
18   // print the table
19   i = 0;
20   while ( i < SIZE ) {
21     printf("%c", tab[i]);
22     i++;
23   }
24   printf("\n");                // print a new line
25
26   return 0;
27 }
```

Listing 4 – Read and write a sequence of characters

10. Write, compile et execute the above program. What kind of data structure did we emulate ?
11. Replace the lines 19 to 24 with printf("%s\n", tab); to use the function printf() to display the content of the array tab as if it was a character string. What happens ? Why ?

```c
#include <stdio.h>

#define TAILLE 4

int main() {
  char tab[TAILLE];
  int i = 0;          /* index allowing us to count the number
                          of read character initalized to 0 */
```

```
9
10    printf("Give %d characters followed by Enter: ", SIZE);
11    while ( i < SIZE ) {  /* loop to read the character (number of
12                             character to read are given by SIZE)
                                 */
13      tab[i] = getchar();
14      i++;                       // incremente the value of i by 1
15    }
16
17    //  tab[i]= 0;
18    // print the table
19    printf("%s\n", tab);
20
21    return 0;
22  }
```

Listing 5 – Read and write a sequence of characters

12. Increase the size of the array by 1 (**char** tab[SIZE+1];) and add the following instruction on line 17 : tab[i]= 0;
13. Recompile and test the program. It should know behave as expected. Explain why.

Remarque. *Pour manipuler des chaînes de caractères correctement, il faut soit connaître le taille, soit être capable de déterminer quand elles se terminent. En C, les chaînes de caractères sont à* zéro terminal*, c'est à dire que le dernier caractère doit avoir comme valeur numérique 0 pour savoir que la chaîne de caractères est finie.*

d) Convert a String to Uppercase

```c
#include <stdio.h>

char upper_case(char c) {
  char m;
  int diff = 'a' - 'A';

  if (c >= 'a' && c <= 'z') {
    m = c - diff;
  } else {
    m = c;
  }

  return m;
}

int main() {
  char c;

  printf("Enter a character: ");
  c = getchar();
  printf("%c\n", upper_case(c));

  return 0;
}
```

Listing 6 – Program `upper_case`

14. Write, compile and execute the program of listing 6
15. What does it do ? Explain line 5 and lines 7 to 11
16. Write a function **int** is_lowercase(**char** c) which returns 0 if c is a lowercase letter and 1 if not ;
17. Modify the program `upper_case` to use the function is_lowercase ;
18. Modify the program to allow the conversion of a whole string to uppercase

Part 3 : Make

a) `Makefile`

`make` is a tool to automatize the compilation of a program. It is particularly useful on huge project where the program is split over many files. Only file which needs to be recompiled are recompiled, and `make` ensure your program is always up to date.

Compilation rules are specified which a file named `makefile` or `Makefile`.

The structure of a rule in a makefile is :

```
target : dependencies list
<TAB>command 1
<TAB>command 2
```

The target correspond to the name of the file generated by the associated commands.

`dependencies list` is the list of dependencies that must be satisfied before running the commands to generate the target. If one of the dependencies are newer that the existing target, the associated commands are being executed. If not, the target is considered up to date and no command is executed.

Remarque. *In a general way :*

— *`dependencies list` might correspond to files or to other make rules*

— *`target` is not necessarily a file. If it is not a file, then the associated commands are always executed*

```
all: hello

hello: hello.c
	gcc hello.c -o hello

clean:
	rm -f hello
```

Listing 7 – Makefile for the program `hello`

In listing 7, we can see the `makefile` allowing to compile the program `hello` if the source file `hello.c` is newer than the target file `hello`.

There are 3 rules :

— the first one is the default rules. If `make` is called without a rule name, the first rule within the `makefile` is being executed. The default rule is usually called `all`

— the second rule describe how to build the program `hello` from the file `hello.c`

— the third one might be used to clean any file produced by a `makefile` rule. It usually called `clean`.

Exercices

1. Open a text editor and write the content of listing 7. Save it to a file name `makefile` in a director named `labC` ;
2. Write the program `hello.c`
3. Execute the command **`make`** within the directory `labC` :

```
$ make
gcc hello.c -o hello
$ ls
hello* hello.c Makefile
```

   We can see that `make` automatically runs the compiler to build `hello` ;
4. Run make again

```
$ make
make: Nothing to be done for 'all'.
```

   Since `hello` is up to date against `hello.c`, `make` did not rerun the compiler
5. Execute the command **`make clean`** :

```
$ make clean
rm -r hello
$ ls
hello.c Makefile
```

6. Run the command `ls` ; What do you notice ?
7. Add a rule in the `makefile` to compile a program called `prog` which should be produced out of the `prog.c` file
8. Should we modify the rule `all` ? Should we modify the rule `clean` ?
9. How could we tel **`make`** to compile only `hello` or `prog` ?

b) Macro

A `makefile` could contain macros. They appears at the beginning of the `makefile` and have the following syntaxe :
`macro_name = value`
We often have the macro `CC = gcc` which specify which C compiler to use. A macro could be called by `$(macro_name)`.
The shell environment variable are accessible within a `makefile` as a macro : the macro `HOME` correspond to the environment variable `HOME`.
Two other macros are commonly used : `CFLAGS` and `LDFLAGS`.
The macro `CFLAGS` allow to specify the options to send to the compiler.
The macro `LDFLAGS` allow to specify the options to send to the linker.

Exercices

10. Modify the `makefile` by replacing the call to `gcc` by a call to the macro `$CC`
11. Add the option `-Wall` to the call to the compiler by using the appropriate macro

c) Special Macros

To simplify the `makefile` and to avoid to rewrite similar rules, `make` provide special macros to write generic rules :

— `$@` : target name

```
prog: prog.c
	$(CC) prog.c -o $@
```

Listing 8 – `makefile` rules with generic target

— `$?` : dependencies list

```
prog: prog.c fonc1.c
	$(CC) $? -o $@
```

Listing 9 – `makefile` with generic dependencies

— Generic rule : % and $* can be used to create generic rules. They allow to write a rules that could be applied on any file name. For example, a generic rule to compile a `.c` file would be :

```
%.o: %.c
	$(CC) -c $*.c
```

Listing 10 – `makefile` with generic rule

Remarque. *You can find more information on how to use `GNU make` at the following URL :* *http://www.gnu.org/software/make/manual/*

Exercices

Let be the following `makefile` :

```
.PHONY: clean

all: test

test: test.o fonc1.o
	gcc test.o fonc1.o -o test

test.o: test.c
	gcc -c -Wall test.c

fonc1.o: fonc1.c
	gcc -c -Wall fonc1.c

clean:
	rm *.o test
```

Listing 11 – `makefile`

12. Use generic rules to simplify the `makefile`

Part 4 : File copy

Write a program which allow to copy a file into an other file.
To do that, you will use the functions `open()`, `read()`, `write()`.
Your program should be composed of multiple function, each of then located in a dedicate source file :

— a function doing the actual copy

— a header file providing the prototype of that functions

— a function `main()`

Compile and run your program on a text file.

Write the correponding `makefile`.

Additional question :
Write a function allowing you to parse command line arguments (source file, detination file). You might have a look at `getopt()` (`man 3 getopt`) for that ;

Part 5 : Annexes

Dec	Hex	Oct	Car	Signification
0	0x00	000	NUL	*Null* (nul, inexistant)
1	0x01	001	SOH	*Start of Header* (début d'en-tête)
2	0x02	002	STX	*Start of Text* (début du texte)
3	0x03	003	ETX	*End of Text* (fin du texte)
4	0x04	004	EOT	*End of Transmission* (fin de transmission)
5	0x05	005	ENQ	*Enquiry (End of Line)* (demande, fin de ligne)
6	0x06	006	ACK	*Acknowledge* (accusé de réception)
7	0x07	007	BEL	*Bell* (caractère d'appel)
8	0x08	010	BS	*Backspace* (espacement arrière)
9	0x09	011	TAB	*Horizontal Tab* (tabulation horizontale)
10	0x0A	012	LF	*Line Feed* (saut de ligne)
11	0x0B	013	VT	*Vertical Tab* (tabulation verticale)
12	0x0C	014	FF	*Form Feed* (saut de page)
13	0x0D	015	CR	*Carriage Return* (retour chariot)
14	0x0E	016	SO	*Shift Out* (fin d'extension)
15	0x0F	017	SI	*Shift In* (démarrage d'extension)
16	0x10	020	DLE	*Data Link Escape*
17	0x11	021	DC1	*Device Control 1 to 4* (DC1 et DC3 sont
18	0x12	022	DC2	généralement utilisés pour coder XON et
19	0x13	023	DC3	XOFF dans un canal de communication
20	0x14	024	DC4	duplex)
21	0x15	025	NAK	*Negative Acknowledge* (accusé de réception négatif)
22	0x16	026	SYN	*Synchronous Idle*
23	0x17	027	ETB	*End of Transmission Block* (fin du bloc de transmission)
24	0x18	030	CAN	*Cancel* (annulation)
25	0x19	031	EM	*End of Medium* (fin de support)
26	0x1A	032	SUB	*Substitute* (substitution)
27	0x1B	033	ESC	*Escape* (échappement)
28	0x1C	034	FS	*File Separator* (séparateur de fichier)
29	0x1D	035	GS	*Group Separator* (séparateur de groupe)
30	0x1E	036	RS	*Record Separator* (séparateur d'enregistrement)
31	0x1F	037	US	*Unit Separator* (séparateur d'unité)
32	0x20	040	SP	*Space* (espace)
33	0x21	041	!	Point d'exclamation
34	0x22	042	"	Guillemet droit
35	0x23	043	#	Dièse ou signe numéro
36	0x24	044	$	Dollar
37	0x25	045	%	Pourcent
38	0x26	046	&	Esperluette ou perluète
39	0x27	047	'	Apostrophe ou guillemet fermant simple ou accent aigu
40	0x28	050	(	Parenthèse ouvrante
41	0x29	051	)	Parenthèse fermante
42	0x2A	052	*	Astérisque
43	0x2B	053	+	Plus
44	0x2C	054	,	Virgule
45	0x2D	055	-	Moins ou tiret ou trait d'union
46	0x2E	056	.	Point
47	0x2F	057	/	*Slash* (barre oblique)
48	0x30	060	0	
49	0x31	061	1	
50	0x32	062	2	
51	0x33	063	3	
52	0x34	064	4	
53	0x35	065	5	
54	0x36	066	6	
55	0x37	067	7	
56	0x38	070	8	
57	0x39	071	9	
58	0x3A	072	:	Deux-points
59	0x3B	073	;	Point-virgule
60	0x3C	074	<	Inférieur
61	0x3D	075	=	Égal
62	0x3E	076	>	Supérieur
63	0x3F	077	?	Point d'interrogation
64	0x40	100	@	Arobase ou A commercial
65	0x41	101	A	
66	0x42	102	B	
67	0x43	103	C	
68	0x44	104	D	
69	0x45	105	E	
70	0x46	106	F	
71	0x47	107	G	
72	0x48	110	H	
73	0x49	111	I	
74	0x4A	112	J	
75	0x4B	113	K	
76	0x4C	114	L	

Dec	Hex	Oct	Car	Signification
77	0x4D	115	M	
78	0x4E	116	N	
79	0x4F	117	O	
80	0x50	120	P	
81	0x51	121	Q	
82	0x52	122	R	
83	0x53	123	S	
84	0x54	124	T	
85	0x55	125	U	
86	0x56	126	V	
87	0x57	127	W	
88	0x58	130	X	
89	0x59	131	Y	
90	0x5A	132	Z	
91	0x5B	133	[	Crochet ouvrant
92	0x5C	134	\	*backslash* ou *antislash* (barre oblique inversée)
93	0x5D	135	]	Crochet fermant
94	0x5E	136	^	Accent circonflexe
95	0x5F	137	_	*underscore* (tiret bas ou souligné)
96	0x60	140	`	Accent grave
97	0x61	141	a	
98	0x62	142	b	
99	0x63	143	c	
100	0x64	144	d	
101	0x65	145	e	
102	0x66	146	f	
103	0x67	147	g	
104	0x68	150	h	
105	0x69	151	i	
106	0x6A	152	j	
107	0x6B	153	k	
108	0x6C	154	l	
109	0x6D	155	m	
110	0x6E	156	n	
111	0x6F	157	o	
112	0x70	160	p	
113	0x71	161	q	
114	0x72	162	r	
115	0x73	163	s	
116	0x74	164	t	
117	0x75	165	u	
118	0x76	166	v	
119	0x77	167	w	
120	0x78	170	x	
121	0x79	171	y	
122	0x7A	172	z	
123	0x7B	173	{	Accolade ouvrante
124	0x7C	174	|	Barre verticale
125	0x7D	175	}	Accolade fermante
126	0x7E	176	~	Tilde
127	0x7F	177	DEL	*Delete* (effacement)

FIGURE 1 – Table de caractère ASCII