2 Basics of C programming

This chapter will serve two purposes. First it will show the basic aspects of using a low level language (like FORTRAN or C) and will introduce the basic features of the C language that will be useful for the rest of the semester.

There are many good manuals and texts to learn C. The classic book is:

- The C Programming Language, Kernighan and Ritchie, Second Edition, Prentice Hall, 1988.

The book is so popular that you should be able to find at least one pdf copy by googling `kernighan and ritchie pdf`. Keep in mind that Ritchie invented C and Kernighan wrote the first C tutorial ever. You should also try to find a good, modern tutorial on the web. In this notes I will follow closely the structure in Kernighan and Ritchie.

Why C and not FORTRAN?

C is widely used outside academia. It is good skill to have if you are interested in having a job in interesting places outside universities and scientific laboratories. Besides, the C compilers are now an standard and are commonly included in any UNIX based operative system.

The reality is that you need to learn at least one low level language in order to be productive and useful in academia. **Actually, if you plan to stay in academia you should learn both**.

But remember that in real life the two most popular programming languages are C and Java [1]. C is estimated to be used by $\sim 18\%$ of the whole programming community. The estimate for FORTRAN is $\sim 0.5\%$

2.1 Write, Compile, Execute

Using a low level programming language implies that there is text that must be written into a file. This file will receive the generic name of **source code**. This file can be edited using any plain text editor. Try to learn EMACS or VIM. In this course we will use EMACS.

2.1.1 Write

Let us first create a working directory called `practice/C/` inside the working directory you have for this class.

```
$ mkdir -p hands_on/C/
$ cd hands_on/C/
```

Now open in emacs a file called `hello.c`

```
$ emacs hello.c &
```

[1] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

There you can type the following piece of source code

```
/*
  Prints the message "Hello World" to the screen.
*/
#include <stdio.h>

main(){
    printf("Hello World\n");
}
```

Save the file.

2.1.2 Compile

In the process of compiling, the **source code** will be translated into another kind of code called **object code** or the **executable**.

In order to compile the code we have written you have to type:

```
$cc hello.c -o hello.x
```

`cc` is a C and C++ compiler. It takes the argument `hello.c`, meaning that it will translate the source code in that file into an executable. The option `-o` followed by `hello.x` indicates that the output executable will be named `hello.x`. If you don't specify any output name, the executable will be called `a.out`

2.1.3 Execute

Run the executable file as follows

```
$./hello.x
```

The result should be

```
Hello World
```

2.2 Arithmetic

Besides printing messages to screen, C can also perform arithmetic computations. Let's start with the following basic example.

```
/*
  Different ways to multiply and divide numbers
*/
#include <stdio.h>

int main(void){
  int a;
```

```
  int b;
  int c;

  float d;
  float e;
  float f;

  a = 1;
  b = 10;
  c = a/b;

  d = 1.0;
  e = 10.0;
  f = d/e;

  printf("%d %d %d \n", a , b, c);
  printf("%f %f %f \n", d , e, f);

  return 0;
}
```

There are other mathematical functions you might want to use. In that case you need to include the header file `math.h`. Take a look at the following example

```
/*
  Using some mathematical operations in math.h
*/
#include <stdio.h>
#include <math.h>
int main(void){
  int a;
  int b;
  int c;

  float d;
  float e;
  float f;

  a = 1;
  b = 10;
  c = pow(a,b);

  d = 1.0;
  e = 10.0;
  f = pow(a,b);
```

```
  printf("%d %d %d \n", a , b, c);
  printf("%f %f %f \n", d , e, f);

  return 0;
}
```

Where the function `pow(a,b)` will compute the expression a^b. You will need now an additional option to compile the code.

```
$cc math_arithmetic.c -lm -o math_arithmetic.x
```

Try to compile it without the `-lm` and see what happens. Also try different numbers for `a` and `b` in `pow(a,b)`.

2.3 Variable types

There are very few basic data types that can be represented by variables in C:

- `char` a single byte, holds a character, i.e. letters.
- `int` integers, the range is limited to the capabilities in the host machine, i.e. you cannot count till infinity in a program.
- `float` single-precision floating point, i.e. real numbers.
- `double` double-precision floating point, i.e. real numbers

Each data type has a different formats to be properly displayed to the screen. Be careful on how the format is assigned to the type. Read, compile and execute the following program to see the differences.

```
/*
Different formats for the different kinds of basic data types in C.
*/
#include <stdio.h>
int main(void){
  char s[100] = "La respuesta es: ";
  int i = 42;
  float x = 42.0;
  double y = 42.0;


  printf("%s %d %f %e\n", s, i, x, y);

  printf("%s %d %d %d\n", s, i, x, y);
  return 0;
}
```

2.4 Printing a table. Loops.

We switch gears to a more complex example where it will become clear how to execute a loop.

```
/*
  Prints a table of values corresponding to
  the radius of a sphere, its surface and volume.
*/

#include <stdio.h>
#define PI 3.14159

int main(void){
  /*defines the variables*/
  int i;
  float radius;
  float volume;
  float surface;

  /*initialization*/
  radius = 0.0;
  volume = 0.0;
  surface = 0.0;

  printf("Radius Surface Volume\n");
  /*loop over 12 different values for the radius*/
  for(i=0; i<12; i++){
    radius = i;
    surface = 4.0 * PI * radius * radius;
    volume = (4.0 / 3.0 ) * radius * radius * radius;
    /*output the values to the screen*/
    printf("%f %f %f\n", radius, surface, volume);
  }

  return 0;
}
```

Here we have two new things. The `#define` statement which can be used to define strings that will be replaced into the source code by the compiler at the moment of compilation. The second novelty is the `for` structure that will be used to make loops, in this case everything starts with the assignment `i=0`, the code inside the curly brackets is executed, at this moment the variable `i` is increased by `+1`, if `i<12` the code inside the curly brackets is executed one more time.

2.5 Arrays

We might want to store **all** the values in the example above. In that case one option can be the following

```
/*
  Example of static array definition and the importance of initalization
*/
#include <stdio.h>

int main(void){
  int lista[10]; //define a list of 10 integers
  int i;

  //print the content
  printf("Content before initilization\n");
  for(i=0;i<10;i++){
    printf("%d\n", lista[i]);
  }

  //initialize
  for(i=0;i<10;i++){
    lista[i] = i * 2;
  }

  //print the new content
  printf("Content after initilization\n");
  for(i=0;i<10;i++){
    printf("%d\n", lista[i]);
  }

  return 0;
}
```

The statement `int lista[10];` is new. It defines an array of ten integers. This is done at the moment of compilation. Read, compile and execute.

Very important points. Array indexing in C starts at `0`, not at `1`. If you need the first three items, you need `lista[0]`, `lista[1]` and `lista[2]`.

Having the arrays in this ways is not very flexible. Many times you don't know in advance how large the array must be. In that case you need dynamic memory allocation.

2.6 Dynamics Memory allocation and pointers

The following is a basic example in memory allocation

```
/*
```

```
  Examples of successful memory allocation and buffer overflow
  i.e. violation of memory safety.
*/
#include <stdio.h>
#include <stdlib.h>

int main(void){
  int i;
  /*future arrays*/
  int *array_int;

  /*number of points in the array*/
  int n_points;
  n_points = 10;

  /*data allocation*/
  array_int = malloc(n_points * sizeof(int));

  /*check if everything went OK*/
  if(!array_int){
    printf("There is something wrong with array int\n");
    exit(1);
  }

  /*print the memory address*/
  printf("Memory starts at %p\n", array_int);

  /*fill the array with data*/
  printf("Allocation went OK. Initializing...\n");
  for(i=0;i<n_points;i++){
    array_int[i] = i*2;
    printf("%d\n", array_int[i]);
  }

  printf("Let's see what happens if I go a bit beyond the allocated space...\n");
  /*What if I try to go a bit beyond?*/
  array_int[n_points] = n_points * 2;
  printf("%d\n", array_int[n_points]);
  printf("OK.");

  /*What if I go far away?*/
  printf("and if I go far away?\n");
  array_int[n_points * 10000] = n_points * 10000 * 2;
  printf("%d\n", array_int[n_points * 10000]);
```

```
  printf("everything went just fine\n");
  return 0;
}
```

The most important thing in this case is the character `*`. This is going to tell you that `array_int` is of type `{int *}`, meaning that it is **a pointer of type int**. The variable `array_int` is going to give you the address of a place in memory.

The `malloc` function will reserve (allocate) a number of bytes in memory. In this case this number is `10` times the bytes that a integer needs to be stored in memory `sizeof(int)`.

The `if` statement check whether the pointer is zero or not. If the value between parenthesis is zero or `NULL`, it is considered as True, and will execute the code between curly braces. From that point on, `array_int[i]` can be used to designate the item `i` in the array.

In C you have a great power to handle memory. But, with great power comes great responsibility. The responsibility is that you have to check that you are never going to use memory outside the region that was allocated.

Read, print and execute the code to see what happens if you try to use memory beyond the bounds of your allocation.

2.7 String Arrays

```
/*
  Basic operations to handle strings
*/

#include <string.h>
#include <stdio.h>

int main(void){
  char name[256];
  char lastname[256];
  char fullname[256];
  int year;


  /*see the garbage in the string before initializing*/
  printf("Garbage in string name %s\n", name);
  printf("Garbage in string lastname %s\n", lastname);


  /*Initialize the string making a copy*/
  strcpy(name, "Simon");
  strcpy(lastname, "El Bobito");
```

```
  printf("After inialization: %s %s\n", name, lastname);

  /*create a string with an special format*/
  year = 1965;
  sprintf(fullname, "%s, %s; Born %d", lastname, name, year);

  printf("Final string: %s\n", fullname);

  return 0;
}
```

2.8 If, do-while

2.9 Your own functions

2.10 Input/Output from files

2.11 Command line input