CS-202 C Bootcamp, week 1: Basics

Important information:

Duration and expected work

This series is scheduled for **the two** sessions of the first week (bearing in mind that you must also assimilate the course content – typically by watching the videos you feel are necessary according to your knowledge/level).

The idea is not to do everything at once, but rather one exercise at a time: the entire series **is not** designed to be finished in 4 hours. The series is more to be considered as a grouping of several exercises, more like a chapter in an exercise book, where each person can *choose* (with the help of the indications below) what they feel is right for them. The **average** level for 2 hours of exercises corresponds to one level-1 exercise and two level-2 exercises.

But, in addition to the 4 hours per week, we also advise you to work on your own at home on a regular basis (personal practice). This extra work should take, on average, to 2 to 3 hours per week. In concrete terms, we would imagine the following workload (for the C bootcamp, i.e. the first three weeks):

- assimilate the content (e.g. watch videos / read a book; ask questions, forum, ...): 3:00 (divided in steps, at home);
- exercises: 4:00 (in steps, at home and in class);
- quiz: 15 min.

Exercise themes and difficulty levels

To help you in your practice (or even your [self-]evaluation), a theme and a level of difficulty (from 1 = easy to 3 = advanced) is associated with each exercise. So give priority to those themes with which you feel least comfortable. Similarly, if you're good at the subject, go straight to level 2 and 3 exercises. If, on the other hand, you're struggling, focus on levels 1 and 2, then come back to level 3 later.

Note that the levels are determined based on when the series is planned. It is clear that the same exercise given later in the year (for example at the time of the exam) would be considered easier!

The final objective of the course is to reach at least level 2. To do this, you must have already faced level 3 several times!

Don't just do the first few exercises, but make sure you've done enough level-2 exercises.

In summary

- **Level 1:** These basic exercises are to be done by everyone in a reasonable time (20 minutes maximum). They allow you to work on the basics.
- **Level 2:** These more advanced exercises **should be tackled by everyone**, without necessarily being finished. Repeating the exercise with the solution should be a good way to progress.
- **Level 3:** These advanced level exercises are for the most motivated/skilled among you. They can **at first** be ignored. **BUT**, they absolutely must be repeated, with the solution if necessary, at the latest during revisions.

Exercise 0: working environment

You will therefore have to write source code in C. This can, of course, be done in any text editor, although it is preferable to have an environment that provides "a little help" (coloring syntax, indentation, compilation, error message handling, debugging, etc.); in short, an "integrated development environment" (IDE).

In this course, we do not impose any tool, but we **strongly recommend** to work on a Linux system (the project will be on Linux anyway). You are free to choose whatever tool you want to write with your C codes. On the VMs of the CO rooms (the only environment on which we can fully answer you), you can use: *Geany* , *vim*, *Emacs*, *VSCodium*, *Gedit*, ...

At a more general level (e.g. your own machine), we can mention: *CodeLite*, *Geany*, *VSCodium*, *Xcode* (for Mac), *CLion*, *NetBeans*, *CDT pour Eclipse*, *Sublime Text*, *kDevelop* (under KDE), *Code::Blocks*, ...
The choice is yours!

Next, you will need a compiler to compile your source codes. Again, it's up to you to choose. At the CO room VM level, `gcc` and `clang` are available.

To summarize, you can work:

- on the VMs of the CO rooms (Ubuntu 22.04; if you are not yet familiar with the connection possibilities to this environment, see this information that I wrote for another course);
- on a Linux machine, physical, virtual or Docker container.

The following explains how to work in an environment similar to that of CO rooms.

To compile your programs:

- as said in the video, on the command line, this is simply done with:

  ```
  gcc -o nom_de_l_executable code_source.c
  ```

- in *Geany*: press F9 (or "Build" in the "Build" menu);
- for a more rigorous version, following a standard:

  ```
  gcc -ansi -Wall -pedantic -o executable code_source.c
  ```

  ```
  gcc -std=c17 -Wall -pedantic -o executable code_source.c
  ```

  In *Geany*: modify the "Build" command preferences ("Set build commands" in the "Build" menu).

To execute:

- in the terminal:

  ```
  ./nom_de_l_executable
  ```

- in *Geany*: press F5 key (or "Execute" in the "Build" menu)

Finally, to help you with the main Unix commands, it might be useful to look at this supplement I wrote for another course.

Exercise 1: Ball bounces (using `for` iteration, level 1)

Objectives

The objective of this exercise is to solve the following problem:

When a ball falls from an initial height ***h***, its speed when it hits the ground is:

$$v = \sqrt{2hg}$$

.

Immediately after the bounce, its speed is *v1=eps*v* (where ***eps*** is a constant and ***v*** the speed before the bounce). It then rises to the height

$$h1 = \frac{v1^2}{2g}$$

.

The goal is to write a program (`rebounds1.c`) which calculates the height to which the ball rises after a number `NBR` of rebounds.

Method

We want to solve this problem, not from a formal point of view (equations) but by **simulation** of the physical system (the ball).

Use a `for` iteration and variables `v`, `v1`, (the speeds before and after the bounce), and `h`, `h1` (the heights at the start of the fall and at the end of the rise).

Tasks

Write the program `rebounds1.c` that displays the height after the specified number of bounces.

Your program should use the **constant** `g`, with a value of 9.81, and ask the user to enter the values of

- **H0** (initial height, constraint: H0 >= 0),

- **eps** (rebound coefficient, constraint 0 <= eps < 1)
- **NBR** (number of bounces, constraint: 0 <= NBR).

Try the values H0 = 10, eps = 0.9, NBR = 20:

```
At the 20th bounce, the height will be 0.147809 m.
```

Remarks:

- To use math functions (like `sqrt()`), add `#include <math.h>` at the beginning of your source file and the `-lm` option to the end of your compile command:

  ```
  gcc monfichier.c -o monfichier -lm
  ```

- In `scanf()`, use `"%d"` to read an integer and `"%lf"` to read a `double`.

Exercise 2: Prime numbers (control structures, level 1)

Write the program `prime.c` that asks the user to enter an integer **n** strictly greater than 1, then decides whether that number is prime or not.

Algorithm

1. Check if the number **n** is even (if yes, it is not prime unless it is 2).
2. For all odd numbers less than or equal to the square root of **n**, check if they divide **n**. If none of them does, then **n** is prime.

Tasks

- If **n** is not prime, your program must display the message:

  ```
  The number is not prime, because it is divisible by D
  ```

  where D is a divisor of n other than 1 and n.

- Otherwise, it will have to display the message:

  ```
  I strongly believe that this number is prime
  ```

Test your program with the numbers: 2, 16, 17, 91, 589, 1001, 1009, 1299827 and 2146654199. Indicate which ones are prime.

Exercise 3: Solving a third-degree equation (variables, arithmetic expressions, conditional connections; level 2)

We now want to make a program which requests three values (a0, a1, a2) from the user and displays the real solution(s) z of the equation:

$$z^3 + a_2\ z^2 + a_1\ z + a_0\ = 0$$

Instructions - start by calculating:

$$Q = \frac{3a_1 - {a_2}^2}{9}$$

$$R = \frac{9a_2a_1 - 27a_0 - 2{a_2}^3}{54}$$

D = $Q^3 + R^2$

Demonstration of the formulas on the page: http://mathworld.wolfram.com/CubicEquation.html

- **If D < 0**, we calculate the three real solutions as follows:

$$\theta \equiv \cos^{-1}\left(\frac{R}{\sqrt{-Q^3}}\right).$$

$\cos^{-1}$ is the `acos()` function of C, provided by `<math.h>`, i.e. you must include this at the start of the program

$z_1 =$ $$2\sqrt{-Q}\cos\left(\frac{\theta}{3}\right) - \tfrac{1}{3}a_2$$

$z_2 =$ $$2\sqrt{-Q}\cos\left(\frac{\theta+2\pi}{3}\right) - \tfrac{1}{3}a_2$$

$z_3 =$ $$2\sqrt{-Q}\cos\left(\frac{\theta+4\pi}{3}\right) - \tfrac{1}{3}a_2.$$

- **Otherwise, we calculate**:

S = $$\sqrt[3]{R+\sqrt{D}}$$

the cube root of x is obtained by "`pow(x,1.0/3.0)`" in C.
Note that the cube root of (`-x`) is the opposite of the cube root of `x`.
We must indeed treat the case where `x < 0` separately from the case `x >= 0`, because C does not accept it in the `pow()` function.

T = $$\sqrt[3]{R-\sqrt{D}},$$

- **If D=0 and S+T != 0**, there are 2 roots:

$z_1 =$ $$-\tfrac{1}{3}a_2 + (S+T)$$

$z_2 =$ $$-\tfrac{1}{3}a_2 - \tfrac{1}{2}(S+T)$$ (double root)

- **Otherwise**, there is a single root: z_1 above.

Exercise 4: Approximate calculation of an integral (functions level 1, then 3)

We can show that for a sufficiently regular function (let's say here C-infinite), we have the following bounds:

$$\left| \int_a^b f(u)\,\mathrm{d}u - \frac{b-a}{840}\left[41f(a) + 216f(\frac{5a+b}{6}) + 27f(\frac{2a+b}{3}) + 272f(\frac{a+b}{2}) + 27f(\frac{a+2b}{3}) + 216f(\frac{a+5b}{6}) + 41f(b)\right]\right| \leq M_8 \cdot (b-a)^9 \frac{541}{315 \cdot (9!) \cdot 2^9}$$

where M_8 is an upper bound of the eighth derivative of f on the segment [a,b].

Write a program that calculates the approximate value of an integral using this formula, i.e. by:

$$\frac{b-a}{840}\left[41f(a) + 216f(\frac{5a+b}{6}) + 27f(\frac{2a+b}{3}) + 272f(\frac{a+b}{2}) + 27f(\frac{a+2b}{3}) + 216f(\frac{a+5b}{6}) + 41f(b)\right]$$

Figure 1: formule

Where the values `a` and `b` are entered by the user.

To do this, write 3 functions:

1. A function `f()` of your choice, which corresponds to the function whose integral you wish to calculate. (Try several cases with x^2, x^3, ..., sin(x), 1/x, etc. You will of course have to recompile the program each time). To use math functions, don't forget to add

   ```
   #include <math.h>
   ```

   at the start of the program and compile with the `-lm` option.

2. An `integrate` function which, from two arguments corresponding to `a` and `b`, calculates the above sum for the function `f()`.

3. A function that asks the user to enter a real number. Use this function to ask the user for the `a` and `b` limits of the integral.

Use these functions in the `main()` to achieve your program.

Note: You can find here a demonstration of the formula given above.

Subsidiary question (Level 3):

How can I avoid recompiling the program for each new function?

As such, letting the user enter their formula themselves is far too complicated for this introductory course.

But if we simplify the problem by giving the possibility to choose from a set of predefined functions (included in the program), then it is feasible using **pointers** to functions (next weeks).

Exercise 5: Exchange (passage by reference, level 1)

Write the program `swap.c` in which you must define (prototype + definition) a function `swap()` which:

- accepts two arguments of type `int*` (passing "by reference");
- exchanges the values of these two arguments.

You will try your function with the following `main()` code:

```
int main(void)
{
    int i = 10;
    int j = 55;

    printf("Before: i=%d and j=%d\n", i, j);
    swap(&i, &j);
    printf("After:  i=%d and j=%d\n", i, j);
    return 0;
}
```

If your function is correct, the program will display:

```
Before: i=10 and j=55
After:  i=55 and j=10
```

Exercise 6: Date stories (control structures, functions, level 1)

[proposed by A. Perrin, 2009]

On December 31, 2008, the 30 GB version of Zune player, an audio player from Microsoft, crashed on startup. The official solution was to wait until January 1st. A bug was discovered in the management of leap years.

Here is the code that was in the bugged version of the Zune player. The number of days since 01/01/1980 (the Microsoft epoch), counting from 1, is recorded in a chip. When it starts, the Zune player reads the number of days and converts it to a date.

```
int days = 10593;       // 31 décembre 2008; comes from the hardware in real
int year = ORIGINYEAR;  // 1980, Microsoft epoch

while (days > 365) {
    if (IsLeapYear(year)) { // 2008 is a leap year
        if (days > 366) {
            days -= 366;
            year += 1;
        }
    }
    else {
        days -= 365;
        year += 1;
    }
}
```

1. Find the error and propose a solution (assume the `IsLeapYear()` function is correct).
2. Implement and verify your solution in a new program named `zune.c` which asks for an integer greater than or equal to 1, representing the number of days since 12/31/1979 (1 corresponds to 01/01/1980 and 10593 to December 31, 2008), then prints the corresponding date to standard output.

   Execution examples:

   ```
   $$ ./zune
   Enter etc.: 10593
   31/12/2008

   $$ ./zune
   Enter etc.: 42
   11/02/1980

   $$ ./zune
   Enter etc.: 1
   01/01/1980

   $$ ./zune
   Enter etc.: 1337
   29/08/1983
   ```

   Clues:

   - write the function `int IsLeapYear(int y)`, which returns 0 if `y` is not a leap year, and a non-zero integer otherwise;
   - write the `DaysForMonth()` function, which returns the number of days in the month given as an argument. The year must also be given as a parameter (why?):

     ```
     int DaysForMonth(int year, int month)
     ```

   - use these two functions to find the date from the number of days since the Microsoft "epoch".
3. To continue on the same theme...

The UNIX "*timestamp*" is the number of seconds since 01/01/1970 at 00:00. Using the `time()` function from the `time.h` library ("man 2 time" `for details), write a`unix-time.c‘ program that displays the current date and time.

Execution examples:

```
$$ ./unix-time
1235583855 secondes se sont ecoulees depuis le 1.1.1970 a minuit.
Nous sommes donc le 25/02/2009 a 17:44:15.
```

Compare your program's release to the current date. Is there a difference? If yes, explain why.

Exercise 7: Multiplication of matrices (arrays, typedef, level 2)

We are trying here to write a program `mulmat.c` which calculates the multiplication of two matrices (recall the algorithm below).

Declarations

- define (`#define`) a constant `N` as the ***maximum*** size for the dimension of a matrix
- in `main()`, declare two matrices `M1` and `M2`.
- Define the `Matrix` type using a structure containing an array and two unsigned integers representing the number of columns and the number of rows.

Functions

- the prototype function

  ```
  Matrix read_matrix(void);
  ```

  which reads from the keyboard the elements of a matrix (after asking for its row and column sizes and checking that these are smaller than `N`) and returns the resulting matrix.

- the prototype function

  ```
  Matrix multiply(const Matrix a, const Matrix b);
  ```

  which multiplies two matrices of compatible sizes and returns the result.

- the prototype function

  ```
  void display_matrix(const Matrix m);
  ```

  which displays the contents of a matrix row by row.

Method

- read from the keyboard the dimensions **l1** (number of rows) and **c1** (number of columns) of the first matrix M1
- read the contents of M1.
- Likewise, read the dimensions then the content of the second matrix M2.
- Check that the number of rows in M2 is identical to the number of columns in M1.
  Otherwise, display an error message "`Matrix multiplication impossible!`"
- Perform matrix multiplication: M = M1*M2:
  - The dimensions of M are: l1 (number of rows) and c2 (number of columns).
  - the element $M_{i,j}$ is defined by
$$M_{i,j} = \sum_{k=1}^{c1} M1_{i,k} \cdot M2_{k,j}$$
- display the result line by line.

Example of use

```
Entering a matrix:
  Number of rows: 2
  Number of columns: 3
  M[1,1]=1
  M[1,2]=2
  M[1,3]=3
  M[2,1]=4
```

```
  M[2,2]=5
  M[2,3]=6
Entering a matrix:
  Number of rows: 3
  Number of columns: 4
  M[1,1]=1
  M[1,2]=2
  M[1,3]=3
  M[1,4]=4
  M[2,1]=5
  M[2,2]=6
  M[2,3]=7
  M[2,4]=8
  M[3,1]=9
  M[3,2]=0
  M[3,3]=1
  M[3,4]=2
Result:
38 14 20 26
83 38 53 68
```

Exercise 8: Complex numbers (structures, level 1)

The goal of this program is to perform basic manipulations on complex numbers: addition, subtraction, multiplication and division.

In the `complexes.c` file, define a `Complexe` structure representing a complex number as two `double` (Cartesian form).

Next, prototype and then define a `display()` function that takes a complex number as an argument and prints it.

In the `main()`, declare and initialize a complex number. Fill it. Compile and run your program to verify that everything works as expected so far.

Prototype then define an `add()` function which takes two complex numbers as arguments and returns their sum.

Also write the functions `subtract()`, `multiply()` and `divide()`.

Test all your functions with the following calculations:

```
(1,0) + (0,1) = (1,1)
(0,1) * (0,1) = (-1,0)
(1,1) * (1,1) = (0,2)
(0,2) / (0,1) = (2,0)
(2,-3) / (1,1) = (-0.5,-2.5)
```

Reminder

the multiplication of `z=(x,y)` by `z'=(x',y')` is the complex number:
`z*z'=(x*x'-y*y', x*y' +y*x')`.
dividing `z=(x,y)` by `z'=(x',y')` is the number complex:
`z/z'=((x*x'+y*y')/(x'*x'+y'*y'), (y*x'-x*y')/(x'*x'+y'*y'))`.

If you need it, here is one reminder more complete on complex numbers.

Exercise 9: Complex numbers revisited (structures, level 2)

We're interested here in solving second-degree equations in the complex field.

Copy the program `complexes.c` into the program `complexes2.c` and edit the latter.

Write the function `Complex sroot(Complex);` which calculates the square root of the positive real part of a complex number.

It can be shown that the square root `z'=(x',y')` of the positive real part of a complex number `z=(x,y)` is given by:

Test before continuing. For example, calculate the square root of -1.

$$x' = \sqrt{\frac{\sqrt{x^2+y^2}+x}{2}}$$

$$y' = \mathrm{sgn}(y) \cdot \sqrt{\frac{\sqrt{x^2+y^2}-x}{2}}$$

où $\mathrm{sgn}(y)$ représente le signe de y,
avec ici la convention $\mathrm{sgn}(0) = 1$.

Figure 2: formule

Declare the `Solutions` type as a structure with 2 `Complex` fields: `z1` and `z2`.

To finish the prototype, define the function `resolve_second_degree()` which takes two `Complex` arguments `b` and `c` and returns, in the form of `Solutions`, the solutions of the equation:
`z*z + b*z + c = 0`

Note: use the same formula as for solving a quadratic equation with real numbers (in the case with 2 solutions), but use your own operators (`multiplication`, `division`, etc.) on the `Complex` to find the `Solutions` of the equation.

Example solutions

With `b=0` and `c=1` we have:

```
z1=-i
z2=i
```

With `b=3-2i` and `c=-5+i` we have:

```
z1=-4.11442+1.76499i
z2=1.11442+0.235013i
```

Exercise 10: Writing to a file (files, level 1)

Write the program `ecriture.c` which:

- reads from the keyboard the **names** and **ages** of different people;
- save this data in a file named `data.dat`.

Your program must also:

- lread values as long as the user does not indicate that he has completed the entry, by pressing the `CTRL+D` keys (which corresponds to the character signaling the *end of file*);
  (**Note:** on Windows, type `CTRL+Z` then `Enter`)
- check that the file was opened correctly, and display an error message otherwise.

Indications:

- remember to check that your extraction (entry) operations are going well;
- don't forget to close the file at the end of write operations;
- and finally, look at the contents of the produced file.

Test your program with the following inputs:

```
Jo      24
Marc    35
Ted     74
Andy     3
Werner  48
OldBob 103
```

Execution example:

```
Enter a name (CTRL+D to finish): Jo
age: 24
Enter a name (CTRL+D to finish): Marc
age: Ted
I'm asking you for an age (positive integer)!
This registration is canceled.
Enter a name (CTRL+D to finish): Marc
age: 35
Enter a name (CTRL+D to finish): ^D
```

Exercise 11: Reading from a file (files, level 1)

In the `lecture.c` file:

1. Display on the screen the contents of the file created during the previous exercise, and also display the number of people contained in this file, as well as the average and maximum ages.
   Check that the file opened successfully, and display an error message otherwise.

2. Then modify your program so that it displays as closely as possible to the following format:

   - name display on 15 characters, *left* alignment;
   - 3-character age display, right-aligned;
   - display of total entries on 2 characters;
   - display of the average on 5 characters with one decimal place;
   - display of minimum and maximum ages like other ages.

Exemple:

```
+-----------------+-----+
| Jo              |  24 |
| Marc            |  35 |
| Ted             |  74 |
| Andy            |   3 |
| Werner          |  48 |
| Bob             | 103 |
+-----------------+-----+
  minimum age     :   3
  maximum age     : 103
6 people, average age: 47.8 years
```

[optional] Exercise 12: binary files (level 1)

In the binary file `a_lire.bin` provided here there are integers (number unspecified) .

1. Write a program that reads the contents of this file and displays it in plain text on the screen.

2. Modify this program so that it "decodes" the previous content by displaying the **character** corresponding to the square root of the number read.

   Pay particular attention to type conversions here, as the function "square root" (`sqrt()` from `math.h`) takes a `double` as argument and returns a `double` (you should get an intelligible message in French).

3. Then write a program to create such files, i.e. a program that reads a sentence from the keyboard and creates a binary file containing an array of integers whose values are the squares of the characters in the sentence (without the final 0 at the end).

4. Testez votre programme avec l'exemple précédemment obtenu.
   Pay particular attention to the signed/unsigned aspect (e.g. with accents).

Note: the file encoded in this way contains an accented character encoded in Latin-1 (ISO 8859-1). This character will be displayed incorrectly on non-Latin-1 *terminals*. This is not a problem, it's not your program that's at fault (just change the terminal encoding).

Exercise 13: Statistics on a file (files, level 2)

Here we want to write a `stat.c` program which calculates statistics on the letters contained in a file.

4.1 Opening the file

Preliminary note: the less confident among you can skip this first part (and the `request_file()` function) and open the file as you prefer, for example as in the previous exercises.
Then come back to this part later when you feel more helpful.

Write a `request_file()` function, which returns `FILE*` corresponding to the open file (or `NULL` otherwise):

```
FILE* request_file();
```

This function, after asking the user to enter the name of the file to read, will open the corresponding file.

In the event of an error when opening, the function will ask again for the name of the file, maximum 3 times. After 3 failures the program will abort and return `NULL`.

Execution example:

```
Name of file to read: stupid.name
 -> ERROR, I cannot read the file "stupid.name"
Name of file to read: stupid2.name
 -> ERROR, I cannot read the file "stupid2.name"
Name of file to read: evenmorestupid.name
 -> ERROR, I cannot read the file "evenmorestupid.name"
=> I give up!
```

Positive example:

```
Name of file to read: stupid.name
 -> ERROR, I cannot read the file "stupid.name"
Name of file to read: data.dat
 -> OK, "data.dat" file opened for reading.
```

4.2 Collection of statistics

Define the `Statistics` type as an array of unsigned long integers.

Write a `initialize_statistics()` function, that takes a `Statistics` variable as an argument (and perhaps other arguments if necessary) and initializes all its elements to 0.

Write a `collect_statistics()` function, prototype:

```
unsigned long int collect_statistics(Statistics to_be_filled, FILE* file_to_read);
```

which collects the number of times each character between space (`' '` or `(char) 32`) and `'ý'` (`(char) 253`) appears in the `file_to_read`.

Thus `to_be_filled[0]` will contain the number of spaces contained in the `file_to_read`, and `to_be_filled[221]` (221=253-32) the number of `'ý'` that `file_to_read` contains.

To read a file character by character, use the function `get(char)`. `man getc` for more details.

The `collect_statistics()` function will return the total number of characters recorded in `to_be_filled`, that is to say the sum of its elements.

4.3 Displaying statistics

Finally write a `display()` function which takes some `Statistics` as argument (plus other arguments if necessary) and displays the statistics collected in absolute (real numbers) and relative (percentages) values.

Absolute values will be displayed right-aligned over 11 characters and percentages right-aligned over 5 characters.

Warning! Characters that did not appear in the file (i.e. having a count of 0) should not be displayed.

Example:

```
STATISTICS:
 :           6 -  12.2%
0:           1 -  2.04%
1:           1 -  2.04%
2:           1 -  2.04%
```

```
3:          3 -  6.12%
4:          3 -  6.12%
...
```