



# B2- Elementary Programming in C

---

B-CPE-200

## Bootstrap

---

Corewar

v1.4



# Bootstrap

## Corewar

---

repository name: CPE\_corewar\_bootstrap\_\$SCOLARYEAR

repository rights: ramassage-tek

language: C

group size: 1

---



- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).



Each exercise is to be submitted in a file called "partX/stepX" in the repository's root, and a Makefile at the root of the directory must compile **all** of the exercises..

The entire libc is allowed for this bootstrap.



If you haven't already done so, please read the Corewar project description.  
If you didn't come to the Corewar Kick-off, well, you should have!

This bootstrap concentrates on the ASM (the compiler that we're asking you to implement in the subject). It aims at helping you, step by step, generating bytecode and creating a small compiler.



# Part 1: Binaries

---

## Step 1

Write a program that creates a file named `./some-text.yolo`. This file must contain "Hello bambino\n" inside of it.

**Binary name:** `./part1/step1/write-some-text`

---

## Step 2

Write a program that creates a file named `./number-as-text.yolo`. This file must contain the character string "12345678".

Your file should be 8 bytes and human-readable.

**Binary name:** `./part1/step2/write-a-number-as-text`

---

## Step 3

Write a program that creates a file named `./number-as-int.yolo`. This file must contain the integer 12345678.

Your file should be 4 bytes and not human-readable.

**Binary name:** `./part1/step3/write-a-number-as-int`

---

## Step 4

Write a program that creates a file named `./several-variables.yolo`. This file must contain 192837 as an int, 'k' as a char and "Corewar is swag!!!" as a char[40].

These variables must be written one after another in the file.

**Binary name:** `./part1/step4/without-padding`

---

## Step 5

Write a program that creates a file named `./one-structure.yolo`. This file must contain 192837 as an int, 'k' as a char and "Corewar is swag!!!" as a char[40].

These variables must form a structure, which must be written in the file.

**Binary name:** `./part1/step5/with-padding`

Note that we're stocking the same information as the previous step, but the file isn't the same size. How come?



# Part 2: Yolo!!!

## The yolotron

Let's invent a new programming language: the yolotron. This language, much more limited than C or even the assembly language proposed in the Corewar project, only tolerates the following instructions:

- add: adds two numbers and displays the result,
- sub: subtracts two numbers and displays the result,
- mul: multiplies two numbers and displays the result,
- put: displays a word.

Here is an example of a super program written in yolotron:

```
Terminal
~/B-CPE-200> cat ./example.yolotron
add 17891 21
add 59 1
sub 21 10
put Yolo!!!!
```

Once this program is compiled and interpreted by the VM, it must display:

```
Terminal
~/B-CPE-200> yolotron-vm ./example.bytecode
17912
60
11
Yolo!!!!
```

If, during compilation, the slightest error is found (for example, too many or not enough parameters passed to an instruction), the "yolotron-asm compiler must display "ahhhh no, don't agree!\n" and terminate.

The compiler must ensure that each line begins with a valid instruction, followed by 2 or 3 parameters (depending on the instruction).

The instruction and its parameters must be separated by one single space. Spaces in the beginning and the end are not tolerated.



## Step 1

Create a program named `yolotron-asm` that can compile yolotron code.

You'll need to translate the instructions and their parameters into binary, using the conversion table:

Instruction	add	sub	mul	put
Hexadecimal Code	0x01	0x02	0x03	0x04

Encoding `put` will be done in the following format `[code instruction][int word length][word]`, which results in  $(5 + \text{strlen}(\text{word}))$  bytes.

Encoding other instructions will be done in the following format `[code instruction][int param1][int param2]`, which results in 9 bytes.

**Binary name:** `./part2/step1/yolotron-asm`

For instance, reusing the previous example:

```
Terminal
~/B-CPE-200> ./yolotron-asm -h
Usage: ./yolotron-asm [source file] [output file]

Terminal
~/B-CPE-200> ./yolotron-asm example.yolotron example.bytecode
~/B-CPE-200> hexdump -C example.bytecode
00000000 01 e3 45 00 00 15 00 00 00 01 3b 00 00 00 01 00 |..E.....;....|
00000010 00 00 02 15 00 00 00 0a 00 00 00 04 09 00 00 00 |.....|
00000020 59 6f 6c 6f 21 21 21 21 21 |Yolo!!!!|
00000029
```

## Step 2

Write a program called `yolotron-vm` that can interpret the instructions of a compiled yolotron program (you have an example higher up in the description).

**Binary name:** `./part2/step2/yolotron-vm`