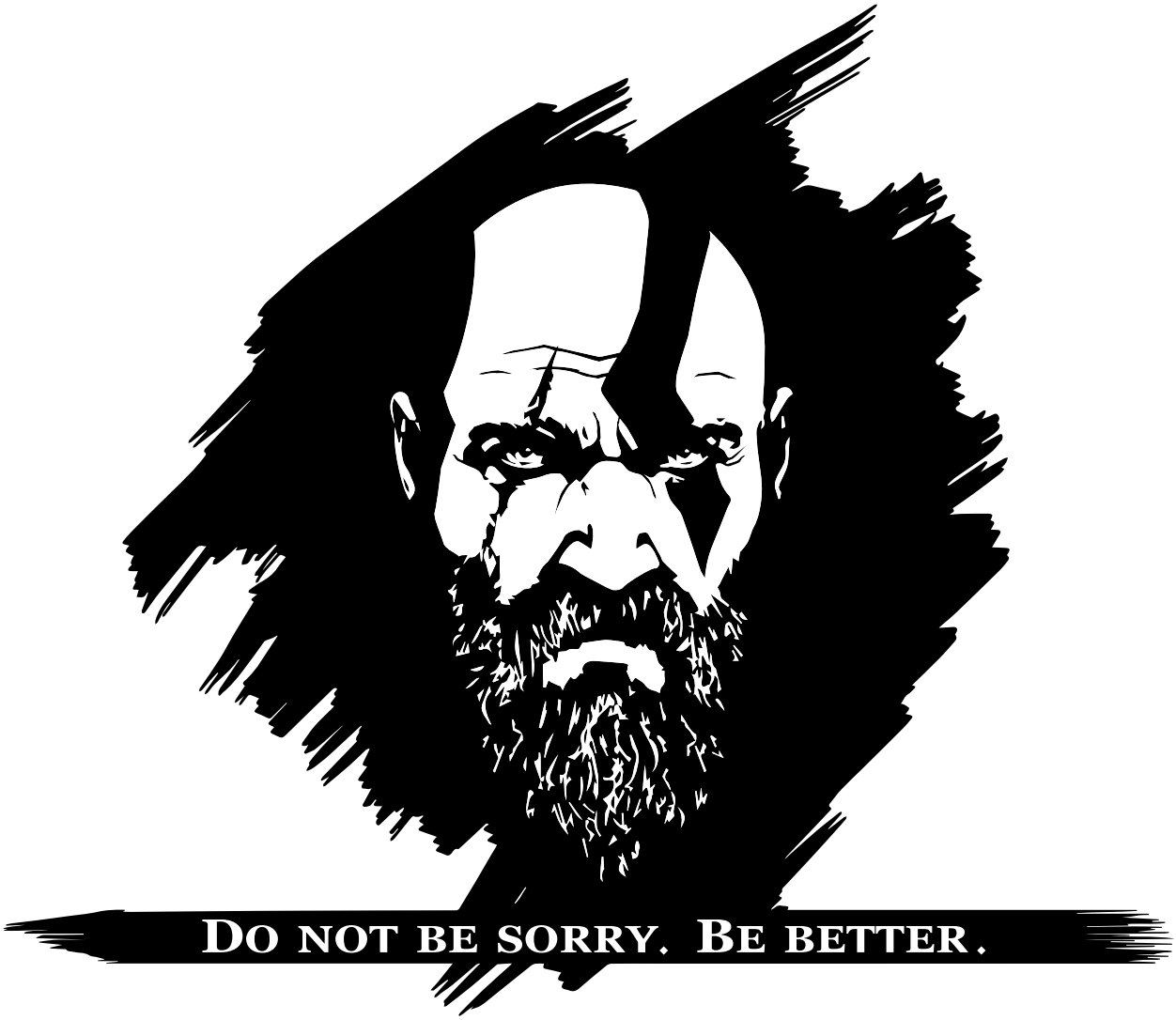
FORMULA ONE — Subject

version **#deploy-formula-one-2021-v0.3**



ASSISTANTS C/UNIX 2019 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Copyright © 2018-2019 Assistants <assistants@tickets.assistants.epita.fr>

The use of this document must abide by the following rules:
- ▷ You downloaded it from the assistants' intranet. *
- ▷ This document is strictly personal and must **not** be passed onto someone else.
- ▷ Non-compliance with these rules can lead to severe sanctions.

Contents

1 Formula One **2**

2 Introduction **3**

3 Viewer **4**

4 Map format **4**

5 API description **5**
5.1 The *update* function . . . 5
5.2 The *move* enumeration . . . 5
5.3 The *car* structure . . . 6
5.4 Other functions . . . 6
5.4.1 *map_get_floor* . . . 6
5.4.2 *map_get_start_** . . . 6
5.4.3 *car_new* . . . 7
5.4.4 *car_clone* . . . 7
5.4.5 *car_delete* . . . 7
5.4.6 *car_move* . . . 7
5.4.7 *vector2_** . . . 7

6 Evaluation **8**

7 Conclusion **8**

1 Formula One

Files to submit:
— `./Makefile`

*. https://intra.assistants.epita.fr

— `./src/control.h`
— `./src/*`
— `./tests/test.c`

Provided files:

— `./src/control.h`
— `./src/formulaone.c`
— `./viewer/libformulaonecontrol.so`
— `./tests/check.c`
— `./tests/check.h`
— `./tests/ghost.c`
— `./tests/utils.c`
— `./tests/utils.h`

Allowed functions: malloc(3), free(3), realloc(3), calloc(3), rand(3), srand(3), time(2)

Allowed includes: math.h

Main function: None

Makefile: Your makefile should define at least the following targets:

— **check**: Produce the *check* binary
— **libformulaone.so**: Produce the *libformulaone.so* binary

2 Introduction

Formula One is all about a thrilling rush of adrenaline flowing through your veins as you go all out on the asphalt. This is exactly the sensations your computer will experiment with this project, thanks to you.

During this project, you have to drive a Formula One car... programmatically. In order to do this, you will drive your car on a map, giving it instructions (such as *accelerate*, *brake*, or *turn left*). You must return a correct sequence of instructions (no crash, and the instructions must lead to the finish line). The speed of your Formula One car is very important: the less instructions you need, the better.

In this project, we want you to focus on the algorithms. Therefore, we give you everything you need to run and test your project:

— A working `Makefile`. You should only have to modify the source files. To build the project, you have to call `make` in the directory which will create a dynamic library used by the viewer.
— A viewer (`viewer`) to watch, in 3D, how your Formula One fares on the asphalt. Everything is done *via* the UI. You can compile and launch the viewer with `make run`
— A `check` executable which allow you to quickly test your AI. You can compile it with `make check`. The executable can be launched with as parameter the map you want to test. It will simply output the number of instructions you used either to reach the finish line or to miserably crash.
— A `ghost` executable which will generate a file containing the moves of your Formula One. You can compile it with `make ghost`. The executable can be launched with the map you want to test your car on as a parameter. It will generate a file containing the sequence of movements returned by your `update` function. If you run your program with a file named `map.frc`, it will generate the file `map.frp`.
— An API (some functions) to move your car (explained in `control.h`)

Here is an example of the procedure to build your project and run the viewer.

```
42sh$ cd path/to/the/formulaone/repository # Go into your repository
42sh$ make             # Build libformulaone.so
42sh$ make run         # Build and launch the viewer with your library
42sh$ make check       # Build the formula one check executable with your library
42sh$ ./check path/to/the/map.frc # Launch check on a map
Crash after 642 steps!
```

You should find everything in the tarball on the intranet. You just have to extract it at the root of your git repository and **rename gitignore to .gitignore**. You do not have to push the viewer on our server. It is just for you to visualize your car.

Be careful !

Do not modify any of the files located in the `viewer` directory nor the `control.h` located in the `src` directory.

They will be overwritten for our tests and will not be tested for the coding style.

The only authorized functions are `malloc(3)`, `free(3)`, `realloc(3)`, `calloc(3)`, `rand(3)`, `srand(3)`, `time(2)` and every function of the standard math library (`math.h`).

The maximum authorized memory consumption is 512Mb. You may use `setrlimit(3)` to ensure you do not use too much memory, **as long as you do not submit it!** Above 512Mb, the behavior of your AI will be undefined.

Formula One is a group project. During the defense, we will check that you have worked together correctly.

3 Viewer

The menu of the viewer allows you to use the following options:

- **New run**: Launch a run. A pop-up will ask you to select a valid map `.frc` file before running your AI.
- **Save a run**: Save a ghost of your AI. A first pop-up will ask you to select a valid map `.frc` file. A second one will ask you where to save the ghost. Ghost files have the `.frp` extension.
- **Play against a ghost**: Launch your AI and a ghost on the same map. A first pop-up will ask you to select a valid ghost `.frp` file. You can launch several ghost at the same time, just select one after each other. Once you have picked all your ghosts, close the pop-up to access a new one where you will be allowed to select a valid map `.frc` file.
- **Quit**: Quit the viewer.

4 Map format

Your car will be tested on maps. A map is composed of map tiles of different kinds. The type of a map tile is given by the `enum floortype`. Possible values are:

- `ROAD`: a road tile. Your car run faster on this kind of tile.
- `GRASS`: a grass tile. Your car can drive on it, but it will be slower.
- `BLOCK`: a block tile. Trying to drive on a block will obviously result in a crash.
- `FINISH`: a piece of the finish line. If your car reaches this kind of tile, the race is over.

The starting point is a special case: your car is directly loaded at the right position (according to the map file), and the corresponding tile is considered to be a `ROAD` tile.

For arbitrary reasons, the initial direction of the car will always be represented by the vector (0, -1) and the angle of this direction will be $\frac{3\pi}{2}$.

The map file format is *very* simple:

- Each tile is represented by an ASCII letter (`r` for `ROAD`, `g` for `GRASS`, `b` for `BLOCK`, `w` for `WATER` and `f` for `FINISH`)
- The starting point is represented by `s`
- There should be the same amount of tiles on each line
- There should be the same number of tiles on each column
- The file extension must be `.frc`

`WATER` will be considered as a `BLOCK` in the API. The only difference with a `BLOCK` will be on the viewer. We will test your function with maps surrounded by block tiles only.

The top-left corner of the first tile of the map is at the position *(0, 0)*, its bottom-right at the position *(1, 1)* and the middle at *(0.5, 0.5)*.

We provide you some maps but you will need to create your own maps to properly test your AI. Submit the maps you will create in the directory *tests/maps/* with a **clear and explicit name** and the correct extension. Their presence will be checked during your defense. As usual, a test suite is also expected.

5 API description

Every detail of the API can be found in the `control.h` file.

5.1 The *update* function

You have to implement the following function:

```
enum move update(struct car *car);
```

This function will be called again and again until your car has either crashed or reached the finish line. Deleting the car results in an undefined behavior (very probably a crash). If you want to keep this car from one call to another, you must **clone** it.

The function must return the next instruction to give to the car.

Be careful !

Beware of floating-point numbers.

5.2 The *move* enumeration

The valid values for the `enum move` are:

- `ACCELERATE`: accelerates your car. New acceleration is `CAR_ACCEL_FACTOR` times the direction.
- `BRAKE`: brakes your car by `CAR_BRAKE_FACTOR`. New speed is `CAR_BRAKE_FACTOR` times the old speed.
- `TURN_LEFT`: turns the car left by `CAR_TURN_ANGLE` degrees.

- TURN_RIGHT: turns the car right by CAR_TURN_ANGLE degrees.
- ACCELERATE_AND_TURN_LEFT: a composition of the ACCELERATE and TURN_LEFT instructions.
- ACCELERATE_AND_TURN_RIGHT: a composition of the ACCELERATE and TURN_RIGHT instructions.
- BRAKE_AND_TURN_LEFT: a composition of the BRAKE and TURN_LEFT instructions.
- BRAKE_AND_TURN_RIGHT: a composition of the BRAKE and TURN_RIGHT instructions.
- DO_NOTHING: no particular action at this step.

Moreover, the car speed is slowed by the CAR_FRICTION_FACTOR (or CAR_GRASS_FRICTION_FACTOR when on grass). The car maximum speed is given by CAR_MAX_SPEED. If your car speed goes below CAR_MIN_SPEED, your car will stop moving.

5.3 The *car* structure

The `struct car` structure has the following members:

- `position`: a 2D vector representing the car location on the map.
- `speed`: a 2D vector of the current speed of the car.
- `acceleration`: a 2D vector of the current acceleration of the car. It depends only on the last instruction.
- `direction`: a 2D vector representing the current direction of the car.
- `direction_angle`: a (counterclockwise) angle. It is the same data as `direction` but represented differently.
- `map`: a pointer to the map the car is currently on.

5.4 Other functions

To further help you focus on the algorithmic part, we also give you these functions to interact with the map and the car.

5.4.1 *map_get_floor*

```
enum floortype map_get_floor(struct map *map, int x, int y);
```

This function returns the type of tile at the position (x, y). If `car` is a pointer to a `struct car`, you could use `map_get_floor(car->map, car->position.x, car->position.y)` to get the type of tile `car` is currently on.

5.4.2 *map_get_start_**

```
float map_get_start_x(struct map *map);
float map_get_start_y(struct map *map);
```

Returns the position of the starting point on the given map.

5.4.3 *car_new*

```
struct car *car_new(struct map *map);
```

Returns a new car (you will have to delete it later). This car is located on its starting position. Its speed and acceleration are the null vector.

5.4.4 *car_clone*

```
struct car *car_clone(struct car *car);
```

Clones the given car. The cloned car must be deleted.

5.4.5 *car_delete*

```
void car_delete(struct car *car);
```

Deletes the given car. The car must have been created by either the `car_new` or the `car_clone` function.

5.4.6 *car_move*

```
enum status car_move(struct car *car, enum move move);
```

Moves the given car according to the given instruction. This function allows you to test what will happen if your `update` function returns the `move` instruction.

The returned value is the result of the instruction (the given `car` is also updated accordingly):

- `NONE`: everything went well.
- `CRASH`: the car has encountered a block.
- `END`: the car has reached the finish line.

5.4.7 *vector2_**

If you want to re-use the `struct vector2` type, you may also use to the following functions:

```
struct vector2 *vector2_new(void);
```

Returns a new null vector (you will have to delete it later).

```
struct vector2 *vector2_clone(struct vector2 *vector);
```

Clones the given vector. The cloned vector must be deleted.

```
void vector2_delete(struct vector2 *vector);
```

Deletes the given vector. It must have been created by either the `vector2_new` or the `vector2_clone` functions.

6 Evaluation

Considering that each instruction you use costs one unit of time, your goal for this project is to pass the finish line with the smallest amount of instructions. As you will use less instructions your car will be faster!

You have to handle maps with block tiles anywhere. **There will always be a valid path from the start to the finish line.** Just find it! Do not forget that the time you take to reach the finish line used will also be tested. Try to go as fast as possible.

At every challenge, your algorithms will be tested against a set of maps (different each time) and their performance will be recorded. We will then establish a ranking based on the number of instructions required to pass the finish line. Bonus points will be given to the top 10 of each race. You will have a total of 4 challenges.

To participate in those challenges, you just have to **push the corresponding tag** given on the intranet like every submission so far.

Before trying to go faster, you should focus on being consistent: we do **not** want a fast car crashing on half the maps ...

7 Conclusion

> *"The more precisely I can drive, the more I enjoy myself."* – Michael Schumacher

Do not be sorry. Be better.