Lab: pointers in the tank plant

1 Objective

Continue the project of *Lab plant*. The program will compute exactly the same results, but the data will travel through pointers, and the readings will be kept in a history allocated in dynamic memory. At the end of this lab you will have:

- passed structures to functions by pointer, using `->` and `const`;
- written functions that receive an array as a pointer and its number of elements;
- returned a second result from a function through an output parameter;
- traversed arrays with pointer arithmetic;
- allocated, used and released a block of dynamic memory with `malloc()` and `free()`, checking for `NULL`.

> Estimated time: 100–120 minutes. Keep following the Barr-C style guide, and fix every warning as if it were an error.

2 Requirements

- *Lab plant* finished: the folder `lab_plant`, which builds without warnings and prints the expected output of that lab. If your version does not work, ask your instructor for the reference version.
- The C file template (`template.h` and `template.c`) of the *C build* session.

3 Starting point

1. Copy the whole *Lab plant* project into a new folder named `lab_pointers`:

```
C:\...> cd %USERPROFILE%\c_projects
C:\...> xcopy lab_plant lab_pointers /E /I
C:\...> cd lab_pointers
C:\...\lab_pointers> mingw32-make clean
C:\...\lab_pointers> del answers.txt
```

2. If you did the optional part of *Lab plant*, remove `tank_update_hold()` from `tank.h`, `tank.c` and `main.c`, so that `main()` uses only `tank_update()`.
3. Build and run with `mingw32-make run`. The output must be the final output of *Lab plant*. Keep it in mind: after Parts 1 and 2 it must not change at all.

4 Part 1: structures by pointer

In *Lab plant*, every call to `tank_update()` copied a whole `tank_config_t` into the function and a whole `tank_state_t` back out of it. Now the functions will receive the **address** of the structures.

1. In `include/tank.h`, change the version to V2.0 in the header comment, and replace the prototypes of `tank_capacity_l()` and `tank_update()` with these:

```
double tank_capacity_l(const tank_config_t *p_config);
void   tank_update(const tank_config_t *p_config, uint16_t raw,
                   tank_state_t *p_state);
```

   `tank_update()` no longer returns a structure. It writes the new state directly into the structure pointed to by `p_state`. Leave `tank_print_status()` as it is: an `enum` is as small as an `int`, so there is no reason to pass it by pointer.

2. In `src/tank.c`, rewrite both functions. Access the fields with the `->` operator (`p_config->height_m` is the same as `(*p_config).height_m`). `tank_update()` does not need a local `tank_state_t` any more: write every field through `p_state`. Update also the `@param` and `@retval` comments.

3. In `main()`, pass the addresses of the elements of the arrays:

```
tank_capacity_l(&plant[tank])
tank_update(&plant[tank], sensor_read_level(tank, cycle),
            &state[tank]);
```

   Build and run. The output must be identical to the one of *Lab plant*.

4. **Try to break `const`.** Add this line temporarily at the beginning of `tank_update()`, and build:

```
p_config->height_m = 0.0;
```

   Keep the message of gcc for question 1, then remove the line.

5. **How many bytes travel.** Add temporarily at the beginning of `main()` a line that prints `sizeof(tank_config_t)`, `sizeof(tank_state_t)` and `sizeof(const tank_config_t *)`. Keep the three values for question 2, then remove the line.

> `sizeof` gives a value of type `size_t`. Its `printf()` format in C99 is `%zu`, for example `printf("%zu\n", sizeof(tank_state_t));`.

5 Part 2: functions that receive arrays

The totals of the plant are still computed inside `main()`. You will move them to a new module, `plant`, whose functions receive the arrays `plant` and `state`.

1. Create the module from the C file template: `include/plant.h` and `src/plant.c`. Rename the include guard to `PLANT_H`, and include `<stdint.h>` and `"tank.h"` in `plant.h`, because its prototypes use the types of the `tank` module:

```
double plant_capacity_l(const tank_config_t *p_plant,
                        uint8_t num_tanks);
double plant_volume_l(const tank_state_t *p_state, uint8_t num_tanks,
                      uint8_t *p_num_faults);
```

2. Write `plant_capacity_l()`. It returns the sum of the capacities of the `num_tanks` tanks, calling `tank_capacity_l()` for each one. `p_plant` points to the first element of the array, so the address of element `tank` is `&p_plant[tank]`, or `p_plant + tank`: both are the same.
3. Write `plant_volume_l()`. It returns the sum of the volumes of the tanks whose sensor is valid, and it writes **through** `p_num_faults` the number of tanks with a faulty sensor. Set `*p_num_faults` to 0 at the beginning of the function.

> To increment the value pointed to by `p_num_faults`, write `(*p_num_faults)++`. Without the parentheses, `*p_num_faults++` increments the **pointer**, not the counter: `++` has higher precedence than `*`.

4. In `main()`, include `plant.h` and use the new functions:
   - Compute the plant capacity with one call: `plant_capacity_l(plant, num_tanks)`. Remember that the name of an array is the address of its first element.
   - At the end of each cycle, compute the plant volume and the number of faults with one call, passing the address of a local variable: `plant_volume_l(state, num_tanks, &num_faults)`.

   Remove the sums and counters that are no longer needed in `main()`. Build and run: the output must still be identical.
5. **The size of an array inside a function.** In *Lab plant* the number of tanks was computed with `sizeof(plant) / sizeof(plant[0])`. Add this line temporarily at the beginning of `plant_capacity_l()`, after the declarations, and build and run:

```
num_tanks = (uint8_t) (sizeof(p_plant) / sizeof(p_plant[0]));
```

Keep the warning of gcc and the plant capacity printed for question 3, then remove the line.

6 Part 3: a history in dynamic memory

The program will now ask how many cycles to run, and it will keep the volume of every valid reading in a **history**. At the end, it will print the minimum, maximum and mean volume of each tank. The size of the history depends on a number typed by the user, so it is not known when the program is compiled: the history is allocated with `malloc()`.

> In a future lab you will save this history to a file. Keep this part well organised: you will reuse it.

1. Create the module `stats` from the C file template, with these functions:

```
double stats_mean(const double *p_data, uint8_t num_elements);
void   stats_min_max(const double *p_data, uint8_t num_elements,
                     double *p_min, double *p_max);
```

   `stats_mean()` is the `mean()` function of the slides. `stats_min_max()` returns two results through `p_min` and `p_max`. Traverse the arrays **with the pointer** (`*p_data`, `p_data++`), without the `[]` operator. If `num_elements` is 0, `stats_mean()` returns 0.0 and `stats_min_max()` writes 0.0 in both results.

2. In `main()`, after printing the configuration, ask for the number of cycles:

```
printf("\nNumber of cycles (1-%u): ",
       (unsigned int) SENSOR_NUM_CYCLES);
fflush(stdout);
```

   Read it with `scanf("%u", &input)` into an `unsigned int` variable. If `scanf()` does not return 1, or the value is not between 1 and `SENSOR_NUM_CYCLES`, print `Invalid number of cycles` and end with `return EXIT_FAILURE;`. Otherwise, store it in a `uint8_t num_cycles`, and make the loop of cycles run `num_cycles` times.

> `fflush(stdout)` forces the prompt to appear before the program waits for the keyboard: the text of a `printf()` without `\n` may stay in a buffer. `EXIT_SUCCESS`, `EXIT_FAILURE`, `malloc()` and `free()` are declared in `<stdlib.h>`.

3. Allocate the history before the loop of cycles: one row of `num_cycles` values of type `double` for each tank.

```
double *p_history;
...
p_history = (double *) malloc((size_t) num_tanks * num_cycles
                              * sizeof(double));
```

If `malloc()` returns `NULL`, print `Not enough memory for the history` and end with `return EXIT_FAILURE;`.

The row of tank `t` starts at `p_history + (t * num_cycles)`:

```
p_history
|
v
+------------------+------------------+-----+------------------+
| row of tank 0    | row of tank 1    | ... | row of tank 3    |
| num_cycles values | num_cycles values |     | num_cycles values |
+------------------+------------------+-----+------------------+
```

4. A faulty reading has no valid volume, so it is not stored. Declare an array `uint8_t num_valid[...]`, with one counter per tank (size it like `state`), and set all its elements to 0 before the loop of cycles. In each cycle, after updating a tank, if its reading is valid:
   - get a pointer `p_row` to the row of the tank;
   - store the volume at `*(p_row + num_valid[tank])`;
   - increment `num_valid[tank]`.
5. After the loop of cycles, print the statistics of each tank, calling `stats_min_max()` and `stats_mean()` with its row and its number of valid readings. Finally, release the history and set the pointer to `NULL`:

```
free(p_history);
p_history = NULL;

return EXIT_SUCCESS;
```

Formats: `"\nVolume history (L), valid readings only\n"` and `"  tank  samples  min  max  mean\n"` for the title and the header, and `"  %4u  %7u  %9.1f  %9.1f  %9.1f\n"` for the rows.

6. Build and run, and type `3`. The end of the output must be:

```
Volume history (L), valid readings only
  tank  samples        min        max       mean
     0        3      460.3      920.6      690.5
     1        3     1294.6     2589.2     1941.9
     2        2      377.1      377.5      377.3
     3        3     6712.8     6904.6     6808.7
```

Run it again with `0`, with `9` and with `abc`: in the three cases the program must print `Invalid number of cycles`. Finally, run it with `6`. The output must be the one of section *Checklist and submission*.

7 Part 4: questions

Answer these questions in a plain text file named `answers.txt`, in the project folder. Short answers are fine, but justify them.

1. What did gcc report in Part 1, step 4, and why? Why does `tank_update()` receive `p_config` as `const` and `p_state` not? In Java, does a `final` parameter that refers to an object prevent the method from modifying that object?
2. Write the three values of Part 1, step 5. How many bytes did each call to `tank_update()` copy in *Lab plant* (argument and returned value), and how many bytes of addresses does it receive now? Is the difference important for 4 tanks? Would it be for 10 000 tanks, or on a microcontroller with 2 KB of RAM?
3. In Part 2, step 5, what value did `num_tanks` take inside the function, and why? What did gcc report, and what plant capacity was printed? Why must `plant_capacity_l()` receive the number of tanks as a parameter? How does Java solve this problem?
4. `plant_volume_l()` produces two results. How does `main()` receive the second one? Replace `(*p_num_faults)++` with `*p_num_faults++`, build and run: what does gcc report, and how many faults are printed in cycle 2? Explain what that line really does. Restore the parentheses afterwards.
5. Why is the history allocated with `malloc()` instead of being declared as `double history[4][6]`? What does the block contain just after `malloc()`? Why must the program check whether `malloc()` returned `NULL`? Why does tank 2 have 5 samples when you run 6 cycles?
6. Remove the line `free(p_history);`, build and run with 6 cycles. Does anything change? Then why is it a bug? When would it become a serious problem? What does Java do instead? Why is it a good habit to set `p_history` to `NULL` after `free()`? Restore the line afterwards.

8 Going further (optional)

1. **Sort the tanks without moving them.** After the statistics, print the tanks sorted by their volume in the last cycle, from the largest to the smallest. Do not move the structures of `state`: declare an array `uint8_t order[...]` with the indices 0, 1, 2, 3, and sort that array with the bubble sort algorithm, using the `swap()` function of the slides adapted to `uint8_t`:

```
static void swap_u8(uint8_t *p_a, uint8_t *p_b);
```

   Write it in the section *Private functions* of `main.c`, with its prototype in *Private function prototypes*. The keyword `static` makes the function visible only inside `main.c`, as the table of `sensor.c` (explained later). With 6 cycles, the program must print:

```
Tanks by volume in the last cycle: 3 0 1 2
```

2. **Keep the last valid value, again.** In *Lab plant*, `tank_update_hold()` needed an extra parameter with the previous state. Change `tank_update()` so that, when the reading is not valid, it changes only `raw`, `sensor_ok` and `status`, and leaves the other fields as they were. Print the numbers also for faulty sensors: in cycle 2, tank 2 must show 0.751 m, 50.1 % and 377.5 L, with status `SENSOR FAULT`. Explain in `answers.txt` why this version does not need the previous state as a parameter, and what you must do with `state` before the first cycle.

9 Checklist and submission

Your lab is complete if:

- ☐ `mingw32-make clean` followed by `mingw32-make` builds the project without any warning.
- ☐ `sensor.h`, `sensor.c`, `area`, `volume`, the `Makefile` and the files in `.vscode` have not been modified.
- ☐ `tank.h`, `tank.c`, `plant.h`, `plant.c`, `stats.h`, `stats.c` and `main.c` follow the C file template, with a comment before every function.
- ☐ No structure is passed or returned by value: the functions of `tank` and `plant` receive pointers.
- ☐ The functions of `stats` traverse the arrays with a pointer, without `[]`.
- ☐ Every `malloc()` has its check for `NULL` and its `free()`.
- ☐ `answers.txt` answers the six questions.
- ☐ With 6 cycles, the program prints the expected output below.

The final output of the program, typing `6`, must be:

```
Plant configuration: 4 tanks
  tank  radius(m)  height(m)  low(%)  high(%)  capacity(L)
     0       0.50       2.00      20       90       1570.8
     1       0.75       3.00      15       85       5301.4
     2       0.40       1.50      25       90        754.0
     3       1.00       2.50      10       95       7854.0
  plant capacity: 15480.2 L

Number of cycles (1-6): 6

Cycle 0
  tank    raw  level(m)  fill(%)  volume(L)  status
     0   1200     0.586     29.3      460.3  NORMAL
     1   2000     1.465     48.8     2589.2  NORMAL
     2   2048     0.750     50.0      377.1  NORMAL
     3   3500     2.137     85.5     6712.8  NORMAL
  plant volume: 10139.4 L (65.5 % of capacity), 0 sensor fault(s)

Cycle 1
  tank    raw  level(m)  fill(%)  volume(L)  status
     0   1800     0.879     44.0      690.5  NORMAL
     1   1500     1.099     36.6     1941.9  NORMAL
     2   2050     0.751     50.1      377.5  NORMAL
     3   3550     2.167     86.7     6808.7  NORMAL
  plant volume: 9818.5 L (63.4 % of capacity), 0 sensor fault(s)

Cycle 2
  tank    raw  level(m)  fill(%)  volume(L)  status
     0   2400     1.172     58.6      920.6  NORMAL
     1   1000     0.733     24.4     1294.6  NORMAL
     2  65535         -         -          -  SENSOR FAULT
     3   3600     2.198     87.9     6904.6  NORMAL
  plant volume: 9119.8 L (58.9 % of capacity), 1 sensor fault(s)

Cycle 3
  tank    raw  level(m)  fill(%)  volume(L)  status
     0   3000     1.465     73.3     1150.8  NORMAL
     1    600     0.440     14.7      776.8  LOW
     2   2047     0.750     50.0      376.9  NORMAL
     3   3700     2.259     90.4     7096.4  NORMAL
  plant volume: 9400.8 L (60.7 % of capacity), 0 sensor fault(s)

Cycle 4
  tank    raw  level(m)  fill(%)  volume(L)  status
     0   3600     1.758     87.9     1380.9  NORMAL
     1    400     0.293      9.8      517.8  LOW
     2   2049     0.751     50.0      377.3  NORMAL
```

```
     3   3850     2.350     94.0     7384.1  NORMAL
  plant volume: 9660.1 L (62.4 % of capacity), 0 sensor fault(s)

Cycle 5
  tank    raw  level(m)  fill(%)  volume(L)  status
     0   3800     1.856     92.8     1457.6  HIGH
     1    300     0.220      7.3      388.4  LOW
     2   2048     0.750     50.0      377.1  NORMAL
     3   3950     2.411     96.5     7575.9  HIGH
  plant volume: 9799.0 L (63.3 % of capacity), 0 sensor fault(s)

Volume history (L), valid readings only
  tank  samples        min        max       mean
     0        6      460.3     1457.6     1010.1
     1        6      388.4     2589.2     1251.5
     2        5      376.9      377.5      377.2
     3        6     6712.8     7575.9     7080.4
```

Submit the folders `src` and `include` and the file `answers.txt`, as indicated by your instructor. Do not submit the `build` folder: run `mingw32-make clean` first. Keep the project: the next labs start from it.

10 Troubleshooting

- `incompatible type for argument 1 of 'tank_update'`**.** The function expects an address and receives a structure. Check the `&` in the call.
- `invalid type argument of '->' (have 'tank_config_t')`**.** You used `->` with a structure, not with a pointer. Use `.` for structures and `->` for pointers to structures.
- `'p_config' is a pointer; did you mean to use '->'?` You used `.` with a pointer. Use `->`, or `(*p_config).field`.
- `unknown type name 'tank_config_t'` **in** `plant.h`**.** `plant.h` must include `tank.h` before its prototypes.
- `implicit declaration of function 'malloc'`, or `'EXIT_FAILURE' undeclared`**.** Include `<stdlib.h>` in `main.c`.
- **The program waits without showing the question.** Add `fflush(stdout);` after the `printf()` of the question.
- **The program crashes, or the statistics show strange numbers.** Check the size given to `malloc()`, and that `p_row` is computed with `num_cycles` and not with `num_tanks`.

- **All the faults are 0, and gcc warned `value computed is not used`.** Read the alert of Part 2, step 3 again.