C Language Level 2 - Exam

January, 18th 2021 - Assignment by Karol Desnos

1. Exam information

1.1. Logistic

- **Assignment organization:** *14* questions on *6* pages.
- **Exam duration:** 2 hours
- **All documents permitted** - **Internet connection, calculators and smartphones are not permitted**
- An archive containing partially pre-filled source-files for this exam will be provided on the Moodle site on the course.
- **Always keep a copy of these original files in case you erase/damage one of them during the exam.**

1.2. Guidelines

Development environment
This exam is prepared for the Linux Lubuntu_Prog Virtual Machine, with the CLion IDE. You may use a classical CLion project or a project with a Makefile, as you prefer. Even if the Makefile is asked as an exercise, you can build your project without it in other questions.

Additional code
If you need to code some other functions than those required, you can do it. As long as you respect the constraints and requirements given in the text, you may code your solution as you want. Nevertheless, there will be penalties if some basic good code rules are not verified.

Testing and commenting your code
Although it is not mandatory, feel free to write tests for your code at the beginning of the main file. In particular, when failing to answer a question entirely, more points will be granted if the limitations of the written code have been identified, and are written down in comments, and/or made explicit with a test.

Indicative time & grading scale.
Given time for each exercise is only indicative. The grading scale is roughly proportional to the time allocated for each exercise. It is assumed that reading the assignment and the given documentation takes approximately 15 minutes.

1.3. Given files

An archive containing partially pre-filled source-files for this exam must be downloaded on the Moodle of the course.

The archive contains the following files:

- Source code: `main.c`, `line.h`, `line.c`, `line_ref.h`
- Pre-built resources: `line_ref.o`
- A pre-configured CLion project `CMakeLists.txt`
- Doxygen documentation for the code: `doxygen` folder generated from given header files. Access to the documentation with `doxygen/html/index.html`.
- An example of DOT file : `example.dot`

Submitting your work

1. Make sure your code compile with no warning or error, and that it is executing without crashing. If your code does not compile, comment the lines responsible for the compilation error. You will lose points if the submitted code has warning or errors or crashes when being executed.
2. Create a directory named `Exam-21-01-LC2-FirstName_LastName`
3. Copy in this directory **ONLY** the header files `*.h`, the C files `*.c`, and the `Makefile` files. **Do not put any binary file.**
4. Check that every file you completed is present.
5. Create a zip archive of the `Exam-21-01-LC2-FirstName_LastName` directory.
6. If necessary, re-name your archive as follows: `Exam-21-01-LC2-FirstName_LastName.zip`
7. Submit your work on the Moodle repository of the course.
8. Make sure the submission was successful.
9. **Do not erase your zip file from your computer**. (It may be useful in case Moodle submission fails).

2. Public Transport Network Map

2.1. Context

The objective of the following exercise is to write a program that builds colorful maps of a public transport networks. A public transport network is typically composed of intertwined bus, metro, and tram lines, each consisting of a list of stations. To help users find their way, a map of such a network must make it as easy a possible to identify where each line goes, and where connections ("*correspondances*" in French) can be made to reach any destination. An example of output produced by the program is shown in Figure 1.

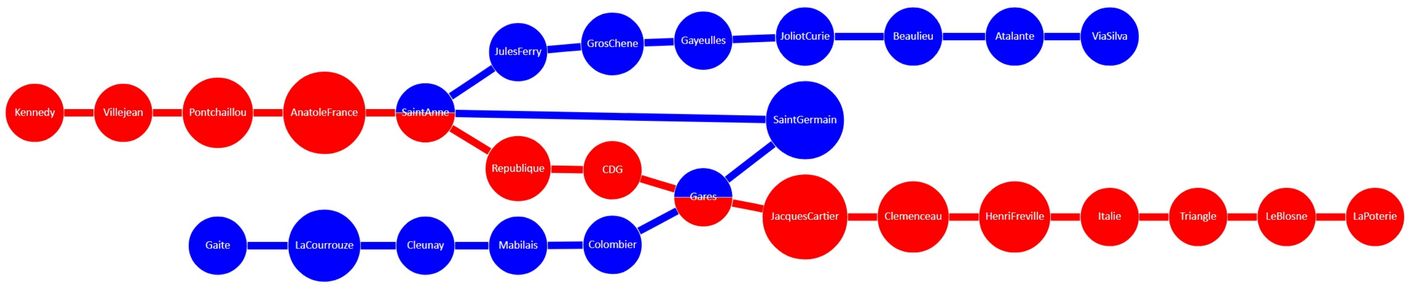

Figure 1 - Rennes Metro Network (click to see the large image)

2.2. Station and Line Structure

The `Line` structure used to represent each line of a network is a doubly-linked list of dynamic length. Each `Line` has its own `name` and `color`. Each element of the `Line` list is a `Station` of this line. In addition to the pointers to `previous` and `next` stations of their `Line`, each `Station` has a `name` and an array of `connections` to other lines. An example of doubly-linked list is illustrated hereafter:

```
Line:

         __________
        |     name-+--------> @
        |    color-+--------> @
        |  current-+---------------.
        |     head-+--.            |
        |__________|  |            |
                      |            |
                .-----'            |
Station:        |                  |
           _____V_____        _____V_____        ___________
          |      name-+->@   |      name-+->@   |      name-+->@
          |      next-+----->|      next-+----->|      next-+----> NULL
  NULL <--+-previous  |<-----+-previous   |<-----+-previous  |
          | connect[] |      | connect[] |      | connect[] |
          |___________|      |___________|      |___________|
```

The C declaration of the `Line` and `Station` structures, from `line.h`, are as follows:

```c
typedef struct Station {
    char *name;               /*!< Name of the station */
    struct Station *next;     /*!< Next station on the line */
    struct Station *previous; /*!< Previous station on the line */
    struct Line *connections[MAX_NB_CONNECTIONS];  /*!< Array of connections */
} Station;

typedef struct Line {
    char *name;               /*!< Name of the line */
    char *color;              /*!< Color of the line */
    struct Station *head; /*!< First Station of the line */
    struct Station *current;  /*!< Current Station of the line */
} Line;
```

It is important to note that in this exam, any properly initialized `Line` should always contain at least one `Station` element. This assumption can be used to avoid checking for list emptiness in all functions used for manipulating the `Line`.

3. Exercises

The 3 exercises are fully independent and can be treated in any order.

- `CMakeLists.txt` : is a fully configured CLion project that can be used to skip exercise 1.
- `line_ref.o` : provides a pre-built implementation of all functions to code in exercise 2. For each function `foo()` to code, the corresponding pre-built function is called `foo_ref()` , as declared in `line_ref.h` .

3.1. Exercise 1: **Makefile (15 min)**

Write the Makefile for the given project. It should provide the three following targets:

Question 1.1.
A `network` target to build the program from the given `line_ref.o` pre-built file and all the `*.c` and `*.h` files of the project. The program built with this target must be usable with debugging facilities of CLion.

Question 1.2.
A `clean` target that deletes every binary file **you** built.

Question 1.3.
As a comment in your Makefile, indicate in a few sentences what the following target does, and why it can be useful? *(Reminder: comments in a Makefile are lines beginning with a* `#` *.)*

```
all: network
    ./network
```

3.2. Exercise 2: **Basic List Features (30 min)**

In this exercise, you are asked to code the basic functions needed for the management of a `Line` , in `line.c` . To get the maximum grade in each question, do not forget to take into account potential corner-cases (eg. `NULL` pointers, failed `malloc` , ...).

We strongly encourage you to test your code immediately after writing it, by calling it from the `main()` function. Even very basic tests can help you detect flaws in your code. To help you visualize the result of each function, you can use the `void printLine(Line *l)` function, provided within `line_ref.o` .

Question 2.1.
Write the `void initLine(Line *l, const char *lineName, const char *color, const char *headName)` function which has the following documentation.

```
/**
 * \brief Initialize the Line.
 *
 * Initialize the line by:
 * - Allocating the needed string an copying the given line name.
 * - Allocating the needed string an copying the given color to it.
 * - Creating a new Station with the given headName (using the createStation() function from line_ref.o).
 * - Setting the line head to this new Station.
 * - Setting the line current station to the head.
 *
 * This function does not allocate memory for the Line structure itself,
 * but only for some of its attributes.
 * Previous attribute values of the Line pointed by the received pointer are ignored.
 *
 * \param[in,out] l the Line to initialize.
 * \param[in] lineName name of the initialized Line.
 * \param[in] color the color code of the Line.
 * \param[in] headName name of the head Station.
 */
```

Question 2.2.
To control movements of the `Station* current` attribute of a `Line` , the following enumeration is defined in `line.h`

```
/**
 * \brief Enumeration used to move along a line.
 *
 * A forward movement will move to the next Station on the line, and a backward
 * movement to the previous.
 * A movement to the head will move the current Station to the Line head.
 */
typedef enum Direction {
    FORWARD,
    BACKWARD,
    HEAD
} Direction;
```

/!\ **Note that the `current` station can be moved to iterate on a line in any function. Saving its original position and restoring it is never required in follow-up questions.** /!\

Write the `int moveCurrent(Line *l, Direction dir)` function which has the following documentation.

```
/**
 * \brief Move the current Station in the given Direction.
 *
 * See Direction comments for more info.
 *
 * \param[in] l the Line on which we move the current Station.
 * \param[in] dir the Direction to use.
 * \return 1 if the move was successful, -1 otherwise.
 */
```

Question 2.3.
Write the `void addStation(Line *l, char *name)` function which has the following documentation.

```
/**
 * \brief Add a new Station at the end of the Line.
 *
 * This function creates and adds a new Station with the given name at
 * the end of the Line, that is, the opposite end to the head.
 *
 * \param[in] l Line to which the Station is added.
 * \param[in] n name of the new Station.
 */
```

Question 2.4.
Write the `Station * getStation(Line *l, char *name)` function which has the following documentation.

```
/**
 * \brief Get a Station with the given name.
 *
 * Return the pointer to the Station with the given name on the Line.
 *
 * \param[in] l the Line within which the Station is searched.
 * \param[in] name the name of the searched Station.
 * \return a pointer to the Station if it is found, a NULL pointer otherwise.
 */
```

3.3. Exercise 3: **Connections Management (30 min)**

In this exercise, you are asked to code the functions needed for the managing the connections of the Stations of a Line in `line.c`. If you are not confident about the proper functioning of functions coded in *Exercice 2*, feel free to use the provided `functionName_ref(...)` alternatives from `line_ref.o`.

Question 3.1.
Write the `int addConnection(Station *s, Line *l)` function which has the following documentation.

```
/**
 * \brief Add a connection to the given Line at the given Station.
 *
 * Add the given Line in the array of connections of the given Station.
 * This function does not add the reciprocal connection in the
 * corresponding Station of Line l.
 *
 * The function does nothing if the given Line is already in the array of
 * connections of the given Station.
 *
 * \param[in,out] s The Station whose array of connections is updated.
 * \param[in] l the Line added to the array of connections.
 * \return the total number of connections of the Station on success,
 *         0 in case the given station was not added to the array,
 *         whatever the cause of failure is.
 */
```

Question 3.2.

Write the `int createConnections(Line *l0, Line *l1)` function which has the following documentation.

```
/**
 * \brief Create all connections between the two lines.
 *
 * This method automatically scans the given lines and fills the connections
 * attribute for the Stations appearing in the two Lines, based on their names.
 * Each connection must have its reciprocal.
 *
 * \param[in,out] l0 the first Line whose connections must be updated.
 * \param[in,out] l1 the second Line whose connections must be updated.
 * \return the number of connection pairs created.
 */
```

Question 3.3.

Write the `int createAllConnections(Line *lines[], int nbLines)` function which has the following documentation.

```
/**
 * \brief Create all connections between all given lines.
 *
 * This function automatically create all the connections between all lines in
 * the given table.
 *
 * \param[in,out] lines array of pointer to the Lines whose connections are created.
 * \param[in] nbLines number of Lines contained in the lines array.
 * \return the total number of connection pairs created.
 */
```

3.4. Exercise 4: **Network Printer (30 min)**

In this last exercise, you will write a generic function `void genericPrint(FILE*, ...)` for generating a customizable text file from a network of `Line`. The coded function is customizable in the sense that it will provide the basic construct to generate some text for each `Line` of the network, and for each `Station` of each `Line`. The generated text itself, for each `Line` and `Station`, will be generated by pre-coded functions, given as function pointers to the `void genericPrint(...)` function.

The complete documentation of the function is as follows:

```
/**
 * \brief Print a customizable text file from a network of lines.
 *
 * The purpose of this function is to print in a file a textual representation of a network of Line.
 * The function will print content in the following order:
 *
 * 1. Header: header string given as a parameter.
 * 2. Iteratively, for each Line l in lines:
 *   1. Line: Call to printLine() for l.
 *   2. Stations: call to printStation() for each Station of Line l.
 * 3. Footer: footer string given as a parameter.
 *
 * \param[in,out] file Pointer to the file where text will be generated.
 * \param[in] header String to print as a header in the generated file.
 * \param[in] footer String to print as a footer in the generated file.
 * \param[in] printLine Pointer to the function used for printing Line info.
 * \param[in] printStation Pointer to the function used for printing Station info.
 * \param[in] lines Array of pointer to the Lines to print.
 * \param[in] nbLines Number of Lines contained in the lines array.
 */
void genericPrint(FILE* file, const char* header, const char* footer,
                  void (*printLine)(FILE*,Line*), void (*printStation)(FILE*,Line*,Station*),
                  Line *lines[], int nbLines);
```

Question 4.1.
Call the `genericPrint(...)` function twice at the end of the `testGenericPrint()` function in `main.c`. Both calls will print the Line from the `lines` array already constructed in the function.

- For the *first* call, use:
  - the standard `stdout` file (no need to open it) for printing content in the console.
  - `NULL` pointers as `header` and `footer` strings,
  - the pre-coded `printLineConsole(...)` and `printStationConsole(...)` functions.
- For the *second* call, use:
  - a properly opened text file named `result.dot`.
  - the pre-coded `dotHeader` and `dotFooter` strings (from `line_ref.o`).
  - the pre-coded `printLineDot(...)` and `printStationDot(...)` functions.

The pre-coded `genericPrint_ref(...)` function can be used for this question. Once a satisfying result is obtained with the `genericPrint_ref(...)` function, replace it with the `genericPrint(...)`, as it will enable you to test your code for the next three questions.

Question 4.2.
Complete the code of the `void genericPrint(...)` function in `line.c` to make it print the `header` and `footer` strings in the given file.

Question 4.3.
Complete the code of the `void genericPrint(...)` function to make it call the `void (*printLine)(FILE*,Line*)` function for each Line.

Question 4.4.
Complete the code of the `void genericPrint(...)` function to make it call the `void (*printStation)(FILE*,Line*, Station*)` function for each Station of each Line.

Visualize the produced DOT file
At the end of this exam, you have generated a `.dot` file that can be converted into an image for visualizing the network of lines.

To produce the graphical version of the map from an `example.dot` file, use the following command in a terminal:

```
dot -Tsvg -Nfontname=Calibri example.dot -o example.svg
```

This will create the `example.svg` vector file, that can be opened with any web browser.