3 Style guide

In this project, you are constrained to use this style guide to maintain your code efficiently. In a random order :

- **Using of compiler flags -Wall -Wextra -Werror -pedantic -pedantic-errors -O3 of g++** to garantee C++ norms in your code implementation, optimizing compiling:
- **Using C++ 11** to use the course ressources. The compiler flag *std=c++11* should be used to do so;
- **Minimizing dynamic memory allocation** : *new*, *new[]*, *delete* et *delete[]*, to optimize security and readability. This will also simplify your code;
- **Dynamic memory using *STL* containers**;
- **Variable naming accouding to their usage**, to minimize redundancy in comments;
- **Perfect indenting of your code** to garantee lisibility and avoid implementation errors;
- **Minimizing duplication of your code** to avoid copypasting and huge maintenance issues that can be inferred from this;
- **Using *snake_case***, with method naming in *object_verb*, variables in *object_noun*;
- **New type should start with a capital letter**;
- **Constants declared in *define* in full capitals**; also known as *SCREAMING_SNAKE_CASE*;
- **Class members called using *this*** to find easily where class members are used and modified;
- **Using *const* where input should be constant**, to garantee read/write correct permissions;
- **Using reference or pointer when passing an argument**.

7 Roadmap

A roadmap is proposed, to represent the various tasks in this project. For this course, it is optional to follow this map, even if it's a good advice :

1. Instanciate a derived class : make a new file, include the parent class and create a class who inherits from `Generic_station_parser`.
   Compile your class file. This should generate zero error, even with the guidelines in the style guide.
2. Make a file with a main function, and instanciate an object of your class. Does it compile? Why?
3. Override `Generic_station_parser::read_stations` in your class : write the prototype of this `protected` method, with the appendix `override`. Implement it.
4. In your main function, instanciate your class and call `stations` method from a `Grade` object. Use the correct one according to your database.

   At this point, you should have finish your first TP.

5. Change your class to make it inherit from `Generic_connection_parser`.
   Does it compile? Why?
6. Implement `read_connections`.
7. Instanciate your class and call `connections` method from `Grade`.

   At this point, you should have finish your second TP.

8. Change your class to make it inherit from `Generic_mapper`.
   Does it compile? Why?
9. Implement `compute_travel`, using station identifiers as nodes. This method should return a vector of `std::pair`, with the station identifier with the current cost as depicted.
10. Implement `compute_and_display_travel`, to display the best path to go from your first station to the other. The path to follow should be displayed in a readable fashion : anyone should be able to follow the inscructions like they are five.
11. Instanciate your class and call `dijkstra` method from `Grade`, passing `false` as the argument.

    At this point you have access to the twenty first points (of twenty) granted in this project.

12. Implement the same last method, using the station names instead of identifiers. You should be error-proof (Type case, syntax, etc) when the user type the station names.
13. Instanciate your class and call `dijkstra` method from `Grade`, passing `true` as the argument.