Command Line Morse Code Converter: A Tool for Encoding and Decoding at Your Disposal

Mentor: Hoàng Đức Chinh

Members: Vương Việt Hùng, Nguyễn Tiến Hưng, Trần Huy Khánh

Abstract

Morse code is a telecommunication method that encodes text characters as sequences of two different signal called dots (.) and dashes (-). Although most prevalent from late 19th to early 20th century, Morse code still retains its fair share of users nowadays due to its simplicity in transmission method, most notably in sending out emergency signal.

The command line tool we developed demonstrates a limited usage scope of Morse code. By converting Morse code text file into plain text and vice versa, our main goal is to help user to get a better understanding on how Morse code is encoded and decoded, while still providing user with a range of options to get some practical uses out of the program.

The mini-project is currently hosted on GitHub. For the project's source code and manuscript, visit our public repository at https://github.com/VuongVietHung156/Morse-Code-Converter.git.

I. Introduction

THE command line Morse Code Converter is a tool that translate a text file written in Morse code to one written in plain text and vice versa. Numerous additional commands are provided that specify special tasks user want the program to perform, namely read the input file as Morse code, read the input file as plain text, print out help message or log conversion information.

The language used in writing the program was C++ for its diverse support of file and string process. Most notably, C++ has a rather similar syntax between writing character to console and to file, this conveniently reduce workload since on multiple occasions, the program will be asked to write information onto two such locations.

We selected Git as our version control system in the need of a tool that help the work flows steady in a team project. There are no actual prevalent reasons for the selection of Git instead of other version control tool other than its popularity and members have been comfortably using it prior to the project.

Other technical aspects will be discussed more in details in the following sections.

*To avoid confusion, project file name, source code as well as command line arguments appearing in this report will be denoted by **bold italic**.*

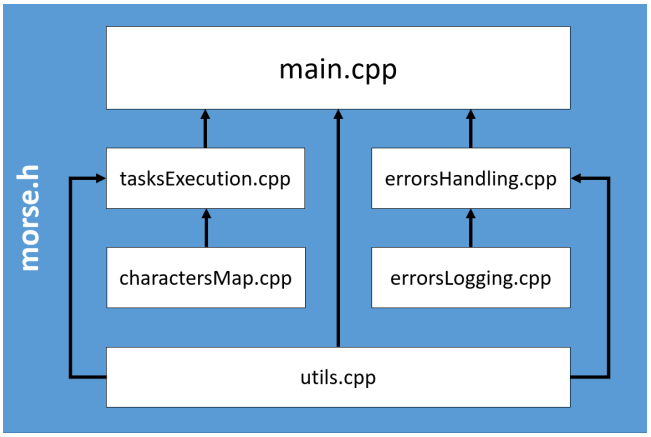
II. Project Structure

The difficult in finding a proper structure to our program in general is hard to ignore. Problem arose as the project itself is miniscule in scale and parts are more often or not, tightly integrated with each other.

Another challenge we had to deal with, is that in nature, command line tools often give user too much freedom in entering inputs, so setting constraint on what user are tempted to enter is both a complicated and essential task in preventing the program from dealing with intelligible requirements.

In the wake of such difficulties in software design. We find the separation of handling user's input from the actual process of perform tasks accordingly is a reasonable, though a rather odd choice in structuring our project.

Figure 1. The structure of the project



As seen from the diagram, the program was designed in a top-down approach, using multiple modules. The arrow symbol (→) represents the inclusion relationship, where the module it points to use functionality from the source module.

Shared functions and variables are declared in the ***morse.h*** header file as well as needed libraries.

A. *main.cpp*

The ***main.cpp*** module is the soul of the program. It gives reader the overhead view of programs' main sections. The module is where the functions which have the highest level of abstraction is called and have two distinct sections:

- Input handle

- Tasks execution

B. ***errorsHandling.cpp***

The ***errorsHandling.cpp*** module makes sure users' input errors are detected and raises the ***bool areInputErrors*** flag to terminate the program.

The input handle process should also make a distinction between file names and command from the arguments list, which should facilitate the proceeding process of tasks execution.

C. ***errorsLogging.cpp***

The ***errorsLogging,cpp*** module's role are the most specialized. Written solely to log the error messages to the console, the module was separated to keep the consistency of error messages.

Functions defined in this module have similar template, they take the source of error and error code as arguments.

D. ***tasksExecution.cpp***

The ***taskExection.cpp*** module perform the heavy lifting tasks. As the user's input bypasses the input condition guard, it will be processed to extracted file names and optional commands. The information will then be used to specify which tasks this module is asked to perform, that includes:

- Deduce type of a file and perform translation (***void convert()***).
- Read the file as morse code and perform translation (***void convertMorse()***).
- Read the file as plain text and perform translation (***void convertText()***)
- Write the help message to the console and to a help.txt file (***void help()***)
- Write the log information to the console and to log file (***void log()***)

E. ***charactersMap.cpp***

Needless explanation, this module simply contains a map from Morse code and corresponding Latin characters and symbols and reverse.

Information stores in two maps will then be utilized by the convert functions in the ***tasksExection.cpp*** module. The separation of two modules is from the fact that we want the ***charactersMap.cpp*** to be easily customizable, as new characters can be added to maps to extend the program functionality.

F. ***utils.cpp***

The ***utils.cpp*** module contains functions that perform miscellaneous tasks. The definition of which tasks should be considered is not concrete, since not every helper function should be putted into to avoid polluting the module.

A good rule of thumb is whether you can immediately associate the helper function with a role performed by the program. A more pragmatic approach would be choosing the functions that are shared between multiple modules.

G. ***morse.h***

The header file is where shared functions and variables are defined. It also contains preprocessor directives, though some of them seem to be closely tied to one specific module. Our decision to keep them collected serves the sake of consistency, since many libraries aren't that specifically pointed to any particular modules but can be utilized by many.

III. Program Demonstration

In this section, we seek to demonstrate in detail how program deal with different inputs and subjects of translation. A more concise version can be found in the README.md file on our GitHub repository. Here, we will explain how program works case by case and give necessary comment on how we designed such behaviors.

A. Input handle

The program input comes in the form of:

./morse [inputfile] [outputfile] [options]

- ***./morse:*** The program name
  User should provide at least one argument proceed ./morse, or else the ***NO_ARGS_PROVIDED*** error will be raised.
- ***[options]:*** Optional commands that specify the task the program is asked to perform.
  A valid command comes in the form of a hyphen (-) followed by a character.
  Had a command of correct format but not included in the list of program command ('-m', '-t', '-h', '-c'), the ***UNRECOGNIZED_CMDS*** error will be raised.
  If both '-m' and '-t' command is provided, the ***CONFLICTED_CMDS*** error will be raised. For obvious reason, you can't treat a file's content both as Morse code and plain text at the same time
- ***[inputfile]:*** Name of the input file – the file that user wish to translate. The first argument to not be recognized as a valid command will be treated as the input file's name.
- ***[outputfile]:*** Name of the output file – the file that store the translation result. The second argument to not be recognized as a valid command will be treated as the output file's name.
  Had the user specify the command '-h' only, then no file names are asked to be provide. Had they not, then two file names arguments should be provided, or else the ***FILE_NAME_MISSING*** error is raised.
  As two first arguments not recognized as a command is treated as file names, then if more than two arguments of such type are provided, it will raised the **INVALID_CMDS** error.
  From this, it is discernable that user should avoid naming their file in the hyphen (-) + character format.

Finally, one should not provide more than four arguments to the program, or else the **TOO_MANY_ARGS** error will be raised. The program gives user freedom to swap place of argument as long as sufficient number of arguments are provided. User

./morse inFile.txt -t -c outFile.txt

should, however, place two file names and commands next to each other. For the sake of readability, such input is not encouraged, though completely valid:

After the input is validate, it will be process to extract file names and commands list with the help of ***void getFileNames()*** and ***void getCommands()*** function.

B. Tasks execution

Before moving on to the tasks execution section. We need to check the file existence status:

- If the input file doesn't exist, the ***FILE_NOT_EXIST*** error will be raised, and the program terminate.
- If the output file has already existed, user will be prompted to overwrite the existing file. Had they refused, the ***FILE_EXIST*** error will be raised.

The checking process of course will be skipped if only the user only asks for a help message.

In case neither '-m' or '-t' command is provided. The program will have to deduce type of the input file base on its content. Had the file contained Morse code character exclusively ('.', '-' and whitespace characters), then it should be interpreted as such, otherwise it's treated as text file.

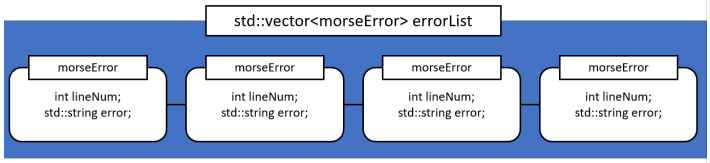
The upcoming section includes multiple control flow statements that will call tasks performing function defined in the ***tasksExecution.cpp*** module to execute user requirements.

A peak inside the tasksExecution.cpp module

The function defined inside the ***tasksExecution.cpp*** module is of high complexity, so a peak into the module is necessary. The module contains several global variables that are used to keep track of conversion process information. The information acquired will then be used to support the logging process.

Also in this module is a notable use of data structure. Here, "error parameters" – the character that is the source of error and the number of line on which it occurs – are stored in the ***morseError*** and ***textError*** struct. As their name suggests, the struct are used to stores error parameters in the conversion processes of Morse code and plain text respectively. Each struct is then stored inside ***std::vector***s. Being dynamically sized, ***std::vector*** extends as the number of error grows, with each element can be easily accessed via indexing in the logging process.

FIGURE 2. THE 'ERROR LIST' DATA STRUCTURE



For Morse code, the definition for what a character is ambiguous. Should each dot and dash be considered as one, or only a whole string of Morse code that can be translated to a Latin character or symbol should be called a "character". We chose to stick with the former definition and using the ***std::string morseCode*** variable name to denote the second notation.

The problem isn't stop at semantic level though, processing Morse code is proof to be much more challenging than that of plain text. That is the reason why a whole function of ***void handleMorse*** is dedicated to process every single piece of Morse code. The inclusion of this function help us reduce the level of nested in the already complex Morse code conversion code.

IV. TEAM WORKING

In this section, we will explain how the project is coordinated as well as how tasks are divided amongst team members.

A. Planning

As the project commence, team members have to present their vision on how the project is going to be structured. Initial idea of separate each task into different module was rejected, because although it arguably helps the work between team members to flows more fluid, the fact that it requires plethora modules just add to the project's complication. We finally settle down with the input handle – tasks execution approach, and the project is then carried out.

B. Team working

As stated previously, our version control tool of choice is Git, and the project is coordinate on one of the members' accounts (the coordinator). The coordinator checks for conflicts on to be merged branches and give comment on other members (the contributor) codes. The detail of how the workload is divided is documented in the table below. Note that only major tasks are denoted.

TABLE 1. TASKS DIVISION

Member	Task
Vương Việt Hùng	Planning Project coordinator Report User experiences • ***errorsHandling.cpp*** • ***errorsLogging.cpp*** • ***void help()*** • ***README.md***
Nguyễn Tiến Hưng	Planning Final version of project structure Program testing Overall project structure • ***main.cpp*** • ***morse.h***
Trần Huy Khánh	Planning Program's main tasks • ***void convertMorse()*** • ***void convertText()*** • ***void logFile()*** • ***void logConsoleInvalid()*** • **void logConsoleUnrecognized()**

IV. Evaluation

Superficially, the program seems mundane, but meticulous details underlying pose serious challenge for designers.

A. Positive remarks

- The separation of input process from tasks execution helps us handle user's input well and profusely. Add a side effect, input process also acts as a tester for tasks execution process, which is arguably more complicated.
- The use of separate namespaces for each high-level function help user understand the workflow of the program just by looking at the main function.
- Our knowledge of data structure comes in handy when the need for storing and accessing complex data arises.
- The use of #enum gives flexibility in adding more error code to the program and was found as a great alternative to the initial use of identifier string.

B. Negative remarks

- Convert functions are found to have extensive use of nested branch statements. This is arguably the soft spot in our software design and is a result of "patching" approach in programming. Had there not been the presence of time pressure, simple refactorization will improve code readability significantly.
- Console logging functions should exist in the ***errorsLogging.cpp*** module for consistency.
- The feature that caused dispute amongst team members is whether '*' and '#' should be treated as Morse code character (since they are used as errors' substitution). We finally decide to stick to the requirement and choose not to, though it could have been a reasonable inclusion.
- The program biggest shortcoming is the inability to overwrite the input file, since this can frequently be the intend of user. We refrain from implement such feature since it requires the program to store input file's content, which can be complicated and for file exponentially large, cause short of memory.

Overall, the team is satisfied with our result, though major improvement in code readability should have been made.

IV. Conclusion

Although only a command line tool, having a diverse range of acceptable input make the program be easily extend for future project, such as become a back-end to a GUI application.

Working with this project gives us valuable experience, it exposed shortcoming in our knowledge as well as our teamwork skill. What we have learnt is multi-file compilation, the use of namespace and header file, the convenient C++ library of ***chrono*** as well as the diversity of C++ standard libraries in general. It also helps us to improve our skill of working with version control tool such as Git.

A project seems simple at first can be extremely complicated once you start programming. As many members working on a same project, we came to a realization on the importance of 'clean code' and having a proper project structure.