graph_alg_test_env

Modularization insturctions

April 14, 2023

1 About document

This document is a part of the `graph_alg_test_env` project by Botond Ortutay. This document is meant to serve any developers using this project in their own projects. It contain instructions on how to add your own code to this software to expand its functionality.

1.1 About modularity

This software is written with modularity in mind. What this means is, that the software is ready to be expanded by other programmers and follows an easily expandable data structure. The concept is, that there are tools in place to load executable code into the project from this data structure. Therefore other developers following the same data structure and using the same tools will be easily able to modify or update this software to their needs. This document aims to describe the steps on how to do this, so that the software may be easily expandable. This document is **not** a full documentation, it is only a description and guide to these features.

2 Modularization instructions

In order to expand this project using your own code, you need to take the following steps:

2.1 Setting up data structure

All code in this project with the exception of the main executable is located in so called "modules". Each module should focus on one funtionality of the software and contain all related code. If I wanted to add add a functionality to my software to color my graphs in a certain way, I would need to make a colorization module for example. It could contain code files to color the graphs red and green. I would need to create a directory for the new colorization module under `app/modules`.

Statement #1: All modules should be located in their own directory under `app/modules`.

Next I would have to make my new module work together with the rest of the software. In order to do that we need to set up a data structure similar to all other modules in the project. First we need a command header file to act as the sole interface between this module and the command line interface of the program. The header file should declare it's own custom namespace and a funciton called `execute()`. The header file should be called `commands.h`.

Statement #2: All modules should have a header called `commands.h` to act as the sole interface between tha module and the cli.

Statement #3: Each `commands.h` file should declare a unique namespace and an `execute()` function within it.

The `execute()` function declared in `commands.h` should also be defined. This function is used to execute this modules commands so that this module can be used via the cli or other interfaces. In order to define the `execute()` function we must create a new file: `commands.cpp`. `commands.cpp` and `execute()` are described with more detail in 2.2

Statement #4: All modules should have a file called `commands.cpp` that contains the definition for the `execute()` function.

In order to make this new module usable with the cli, it has to be compatible with the `loadModule()` function defined in `app/modules/main/module_loader.cpp`. To do that we need to set up a csv file called `commands.csv` that contains metadata for all commands. The metadata consists of the command itself, a unique int type id, and a short description text of what the command does. `commands.csv` is described with more detail in 2.5.

Statement #5: All modules should have a file called `commands.csv` with `command,id,description` formatted metadata for all commands.

With these files in place the new module now has the correct data structure and the following steps can be taken to integrate the module with the rest of the program.

2.2 About `commands.cpp` and `execute()`

In this section we describe the things you should note when setting up the `commands.cpp` file as described in **Statement #4**. First: the `execute()` function takes an integer as a parameter. This integer is the unique command ID each command has and it's used by the `execute()` to identify which command it needs to execute. This means that there needs to be a decision making process in `execute()` where the function executes the command which has the inputted id.

Statement #6: `execute()` is inputted an integer and executes the command with the ID matching the input

All command IDs of the same module should be of the same range, beacuse command ID validation is handeled based on range. As each module has a maximum of 100 commands, a 100 length integer range must be chosen for the new module. This must not conflict with the range of any other module. This range should be documented.

Statement #7: Each module has a limit of maximum 100 commands

Statement #8: For each module a 100 long command ID range should be chosen. All the command IDs of this module must be on this range.

The commands themselves are defined inside the `execute()` function. They should be made so that the module's functionalities can be used outside the modules via the interface described in 2.3. The commands should contain the functions and code they need to execute and they should return a string to be printed on to the CLI.

Statement #9: Each command should be defined inside the `execute()` function and return a printable string for the CLI.

Note that the `execute()` function gets called from `app/modules/main/command_parser.cpp` with a pre-validated command ID. Therefore command validation should **not** be handeled inside `commands.cpp`. Input validation and setting up `command_parser.cpp` are described in more detail in 2.4.

2.3 Interfacing between modules and CLI

Interfacing between the modules and the CLI happens through the `commands.h` header. In the CLI end all files that have the ability to call commands from modules load the header `app/metaheaders/commands_meta.h` . Therefore making the new module interface with the CLI is as simple as including `commands.h` in `app/metaheaders/commands_meta.h` .

Statement #10: `commands.h` files of all modules should be included in `app/metaheaders/commands_meta.h` .

2.4 Command validation with `command_parser.cpp`

Each modules `execute()` function gets called from `app/modules/main/command_parser.cpp` with a valid command ID. When introducing new modules, this functionality needs to be set up by modifying `command_parser.cpp` . You do not have to worry about function visibility here. If 2.3 was done properly, the `execute()` function of all modules should be visible here. To set up command validation with `command_parser.cpp` correctly, you need to find the part of the file marked as `--- COMMAND PASSTHROUGH ---` and modify the `if - else` structure, so that if `commandId` is in your modules id range, as described in **Statement #8**, `yourModuleNamespace::execute(commandId)` gets returned. It is important to get the namespace right, because all modules have `execute()` functions.

Statement #11: In the `--- COMMAND PASSTHROUGH ---` section of `app/modules/main/command_parser.cpp` : if the commandId variable matches a modules id range, `moduleNamespace::execute(commandId)` should get returned.

You should note, that there is a hardcoded limit of 100 modules total and 100 commands/module. Therefore the program cannot handle more than 100 modules. This limit affects many files and lifting it will require significant changes to the program. Another thing to note is that **from this point on** you will have to compile by hand or change the `Makefile`, because the new files introduced by the new module won't properly compile and link with the old `Makefile`.

2.5 Loading new module with `module_loader.cpp`

The last thing we need to do for the new module to completely integrated with the CLI is to actually be able to load the new commands into the command data system for this we need to create a command data file as described in **Statement #5**. `app/modules/main/module_loader.cpp` then takes the metadata out of this file and inserts it into the command data structure when the `loadModule("moduleName", commDataStructure)` function gets called. Therefore this command should be inserted into `app/main.cpp`.

Statement #12: All modules should be loaded by calling the function `loadModule("moduleName", commDataStructure)` from `app/main.cpp`

2.6 Editing the `Makefile`

If you only ever need to add one module and never touch the code again (and if we can assume that no one else will touch your code again), this step can be skipped. If however you want to continue adding module or editing the code of this program, this step will make your life a lot easier as you won't have to compile, link, test etc. by hand every time. The old `Makefile` can be made to work with the new module if you do the following modifications to it:

1. Compile your new `commands.cpp` file, as well as all new `.cpp` files you want to include in the new program by adding the commands `$(CC) -c app/modules/MODULE/FILE.cpp` in the `all:  # program` section of the `Makefile`
2. This should generate `FILE.o` output files for each compiled file.
3. since the file `commands.cpp` is included in all modules we need to rename all `commands.o` files before the next `commands.cpp` gets compiled, so that no file gets overwritten. therefore the line `mv commands.o UNIQUE-NAME.o` should be added before the line `$(CC) -c app/modules/MY-MODULE/commands.cpp`
4. Under `all:  # program` find the line `$(CC) main.o ...  .o`. Here the `.o` files get linked together to form one big program out of all the files. Add the new `.o` files to the end of this line.
5. To remove all output files after compiling and linking the source files add the line `$(RM) FILE.o` in the `all:  # cleanup` section of the `Makefile`

6. If you made new tests or if your new code somehow affects the old tests, make sure the tests also compile and link correctly.

7. add your new source files to the dependency variables (under `# main program dependencies` and `# additional dependencies for testing`)

8. running the `make` command on your command line should now compile and link everything correctly if done right.