Roost Controller

Fares Elsabbagh

**Introduction**

This document describes how the provided project implements a smart thermostat. The solution is implemented in C++. To accomplish this assignment, three different categories of classes were implemented: category to simulate command input by the user, category to represent different types of commands, and a category to control the environment. The most enjoyable and challenging part of the project was designing a flexible polymorphic implementation that could easily be modified in a real-life situation.

**Implementation**

### **Category 1: Simulating input by the user**

* Class userCommandController
  + userCommandController class imports all the user commands from the scenario file and provides a real-time input stimulator to the roost controller. The imported user commands are divided into two vectors:
    - <allCommands> is a queue of all non-routine related commands, such as independent SET\_TEMP, PRINT\_STATUS, START\_ROUTINE, or RESUME\_ROUTINE.
      * This is only set at the beginning of the program.
    - <routine\_commands> is a queue of the current routine’s commands.
      * This is usually set when a START\_ROUTINE or RESUME\_ROUTINE commands is executed and emptied when the system is not following a routine.
      * <routines> is a vector that stores all the created routines as routine objects. userCommandController uses it to find the routine commands when a START\_ROUTINE or RESUME\_ROUTINE is called.

### **Category 2: Representing different types of commands**

* Class userCommand\_t
  + userCommand\_t is the base class for all the different types of commands.
  + A userCommand\_t object is instantiated only when userCommandController has no commands left for a specific time stamp so it returns a “CONT” command which tells the roost controller to continue.
* Class set\_temp (derived from userCommand\_t class)
  + This class represents a SET\_TEMP command.
  + This SET\_TEMP command holds information regarding how it was created
    - <time> the time the command should be inputted to the system.
    - <command> is the name of the command “SET-TEMP”.
    - <temp> is the target temp for the command or the option to this command.
    - <state> 0 if an independent command 1 if part of a routine.
    - <routine\_name> is the name of the routine that created it.
    - <routine\_created> is the time the routine was created.
* Class print\_command (derived from userCommand\_t class)
  + This class represents a PRINT\_STATUS command.
  + Since there are no options for PRINT\_STATUS the only information needed to create a print\_command is the time stamp and the string “PRINT\_COMMAND”.
* Class routine
  + This class represents a routine and its data. It is NOT derived from userCommand\_t
    - <routine\_created> is the time the routine is created.
    - <routine\_name> is the name of the routine.
      * This is used by userCommandController to find the routine and fetch its commands.
    - <routine\_commands> is a queue with all the SET\_TEMP commands related to that routine.
* Class routine\_command (derived from userCommand\_t class)
  + This class represents commands like START\_ROUTINE and RESUME\_ROUTINE.
  + This command is not actually sent to roost controller, but instead is used by userCommandController to find the correct routine in vector <routines> and sets queue userCommandController.<routine\_commands> equal to routine.<routine\_commands>.

### **Category 3: Controlling the environment**

* Class roostController
  + This class is utilizes all the different classes and does the actual controlling of the environment and command execution.
  + checkDst() function is the main function in the controller.
    - It gets the current time from the environment.
    - It checks with userCommandContoller if there are any commands to be executed at that time.
    - It checks for overheating and makes sure the AC and the Heater have enough time to cool down.
    - It manipulates with the environment by turning on the AC and the heater depending on the target temperature in an efficient way and makes sure both of them are never on at the same time.
  + It houses important data that is needed to achieve its function
    - <envConfig> that has the initial configuration of the environment.
    - Temperature related variables.
      * <currTemp> is the current temp of the environment.
      * <dstTemp> is the target temp for the environment.
      * <actualDstTemp> is the actual target temp used that is sometimes different from <dstTemp> for efficiency reasons.
    - Routine related variables.
      * <command\_state> is whether the environment is being controlled by a routine or independent commands.
      * <routine\_name> is the name of the current routine controlling the environment.
      * <routine\_created> is the time the routine was created.
    - Safety related variables.
      * <acSafeToOpen> is whether it is safe to open the AC.
      * <acSafeClose> is whether or not the AC needs to be closed immediately because of overheating.
      * <heaterSafeToOpen> is whether it is safe to open the heater.
      * <heaterSafeClose> is whether or not the heater needs to be closed immediately because of overheating.

**Implementation Evaluation**

The roostController and the main function can only access the scenario file commands when the current time of the environment is the same as the time of the command, as opposed to them having access to all the scenario file commands at once. This is done to simulate a real-life scenario where the user could input a command at any time and have the flexibility to change the way commands are inputted to the system. For example, userCommandController could be changed in the future to receive commands as interrupts from actual IO devices like a touch screen or a remote controller. This could be accomplished without having to change anything code in roostController or the environment.

If I were to redo the project, I would add a command input method from the terminal which would create an interrupt to be handled by userCommandController. I would also write smaller functions that accomplish smaller tasks rather than big and loaded functions and I would also group related data and variables in structs instead of independent variables.

**Challenges Faced**

One of the biggest challenges faced implementing this project was trying to deal with multi-routine scenario files and how routine commands should be stored. A design decision was made to use vectors to store routines created which would have a queue with all the routine commands. This would lead to a local queue which would hold the current routine’s queue that the user wants to run. Another design challenge was the time constraint as I would have enjoyed implementing more features and writing a more organized code but I was limited by the time when I was not studying for exams or working on projects.