COMP_SCI 211 – Fall Quarter, 2019
Programming Assignment 8

Written by Larry Henschen, Northwestern University

Time Estimate: ~6-9 hours[1]

In this assignment we complete our simulator by adding the scheduling of events, the processing of events by the individual devices, and simulated malfunctions.

Background:

This assignment completes our simulated car system. In Assignments 3 and 6, we implemented devices, made to represent objects within a car, and stored them all in an array named 'devices'. We could use an array to store them all, because there are a finite number of devices per car that are consistent throughout the entire duration of the car. There will always be turn signals, and there will always be a brake. We also don't care about the order of the devices within the array, so we can just add a new device to the next empty cell in the array. In Assignment 7, we implemented the event class, made to represent things that happen to our devices. We used a linked list, named 'eventList', to represent this class because events need to be added or removed at various points in time, depending on what we want to have happen. So, we need a structure that is very flexible. A linked list is perfect since the nodes in the list are not stored in sequential cells in memory. Adding a new node to a linked list is as easy as creating the new node and then adjusting two pointers, regardless of where it belongs in the list.

As of right now, our devices array and events linked list are separate entities. Our task in this assignment is to link them together. Our events will represent changes that need to happen to the devices at various points in time. We want it to be that when sensor devices (analog or digital) change value, events are entered into the list. For example, the pressure on the brake pedal changes and that triggers the creation of a new event. In addition, some events themselves cause new events to be scheduled; in our case, when the turn signal is ON, the turn lamps are supposed to flash. When the lamp goes ON, we must schedule an event for a short time in the future for it to go OFF again and *vice versa*.

In some cases, events that are in the list need to be removed. For example, when the turn signal is ON, there will always be an event scheduled for the corresponding lamp, either to go ON or go OFF. However, when the signal goes OFF, the lamp should go OFF and any event in the list for that lamp should be removed from the list; otherwise, the lamp would just flash forever.

In this project we will face a common occurrence in larger software projects. We will have several variables that represent information common to many operations, like the time. One way to handle this is to include these as parameters in the function calls. We

[1] This assignment is very long, and extremely difficult to debug. Start early

could pass the time in as an input to each function that needs it. But that leads to functions with many arguments, and in many cases the function doesn't really use the information but just passes it through to other functions that it calls. A way around this is to use global variables (i.e. variables declared at the file level in one cpp file) and to arrange for them to be referenced in the other cpp files. The C/C++ attribute that allows such reference to global variables declared in other files is **extern**. For example, in this project we will create a new global variable, checkEngine. This will be declared at the file level in main.cpp:

```
bool checkEngine;
```

To allow this to be referenced in system_utilities.cpp, that file will contain the line

```
extern bool checkEngine;
```

The variable will be created in main.cpp, but we will have access to it throughout all of system_utilities.cpp

Your Task:

1. Add a new variable in main.cpp at the file level, a bool called checkEngine. This variable should be initialized to false.

2. Add extern declarations of the following variables in the following files:

   In system_utilities.cpp:
   ```
   extern device* devices[MAX_DEVICES];
   extern int numDevices;
   extern int globalTime;
   extern LIST* eventList;
   extern bool checkEngine;
   ```

   In devices.cpp:
   ```
   extern int globalTime;
   extern LIST* eventList;
   extern bool checkEngine;
   ```

3. Implement the following function in system_utilities.cpp (prototype is provided in system_utilities.h).

   int **findDevice**(string d) - The argument d is the name of a device. This function searches the array of devices for a device whose name matches d. If found, this function returns the index of where that device is located. If not found, this function returns -1. You previously wrote a function that compares the name of a device to a given string. You should use that function.

4. Add the following member function to the LIST class (it is already in events.h, your job is to implement it in events.cpp):

   void **removeEventsNamed**(string n) - Remove any events whose device name matches n. There may be multiple events in the list that have the same name. You must remove them all.[2]

5. Notice that I have moved the body of the process events case (of the main.cpp switch) into a function called **processEvents** in system_utilities. Modify the **processEvents** function as follows.

   Look in the comments as to where your code will go. Compare the name of the event you're currently on with each of the following device names (below). If the name matches, call the appropriate process function. If the device name does not match any of the following, do nothing and go on to the next event. Each of the process functions itself will have the form:

   void processXYZ(EVENT* e);

   where e is the pointer to the EVENT. See prototypes for these functions in system_utilities.h. The behavior of each of these functions is specified here. In addition to the changes to the processEvents function, you also need to define these 'process' functions in system_utilites.cpp.

   Please read through each behavior carefully, they are not all consistent. Closely read through Brake and Accelerator; many students have gotten these wrong in the past by not reading them carefully.

   - Left Turn Signal - If the value in the event is ON, find the "Left Turn Lamp" device (in the devices array), set its value to be ON, then create a new pointer to an event on the heap to turn it OFF at the current (global) time+2 and insert that new event pointer into the event list. You should use the event constructor here. The name of the new event is "Left Turn Lamp". If the event value is OFF, remove events with name "Left Turn Lamp" from the event list, find the "Left Turn Lamp" device (in the devices array), and set its value to be OFF.
   - Right Turn Signal - Similar, but using "Right Turn Lamp".
   - Brake - Find the "Brake Lamps" device (in the devices array). If the event value is ON, turn the "Brake Lamps" device ON, otherwise turn the "Brake Lamps" device OFF.
   - Accelerator - Find the Motor device (in the devices array). Have the Motor device set its value to the value in the event object.
   - Left Turn Lamp - Find the "Left Turn Lamp" device (in the devices array). Set its value to the value in the event object, then schedule a new event for "Left Turn Lamp" to switch to the opposite value at current time + 2. Remember that the value will be one of ON or OFF.

[2] This function is as logically complex as insertEvent. Beware!!

- Right Turn Lamp - Similar, but using "Right Turn Lamp".
- Speedometer – Since the speedometer doesn't have an impact on any of the controllers, this function will be empty (literally {}).
- Motor - Since the motor also doesn't have direct impact on any of the controllers, this function will be empty (literally {}).

6. In the TIME_CLICK case of the main switch, after incrementing the time, call the **processEvents** function.

7. When the value of a sensor (either analog or digital) changes, this will trigger the creation of an event. For example, in one moment, if the Left Turn Signal becomes ON, then an event is triggered so that the Left Turn Lamp can flash ON and OFF in the future. To enable this behavior, make the following changes to the device class and its derived classes:

   - Add a new member function to the device class: void **createEvent**(int tm). This function should create a new pointer to an EVENT on the heap with time tm, value the current value of this device, and device name the name of this device. Then insert the new EVENT pointer into the event list. As always, this is already in devices.h. Your job is to implement it in devices.cpp.
   - In the digital and analog sensor **setValue** functions (after you set the value), if the current time is greater than 0, create a new event at the current (global) time and add it to the event list (notice that both of these things are accomplished by the **createEvent** function). You should **not** do this for the digital and analog controller devices. NOTE: In our project, all the devices will be created at the beginning. The condition that the time be greater than 0 means that the devices won't cause events when they are first created.

     In the device class, notice that **setValue**() is now a pure virtual function. As with any virtual function, this will allow the SET_DEVICE_VALUE case of the main switch to have any element of the devices array call its set value function. It is <u>pure</u> virtual because it will not be implemented at all in the device class. (By the way, that means you would not be able to create a device object as it is now an abstract base class, but in our project, we only create objects of the four derived types.)

8. Implement the SET_DEVICE_VALUE case of the main switch. The additional tokens on the line are the device name and the new value. Find the device in the devices array, and have that device set its value to the one read from the command line. If the device is a digital sensor or digital controller device, the value from the input file will only be denoted by the numbers 1 or 0, which you must translate to the constants ON or OFF respectively. For analog sensor and analog controller devices, the value token can be denoted by any possible number (including potentially 1 or 0) Analog sensor and analog controller values have no special meaning and are purely numerical (i.e. 1 is the int 1, not ON). Remember to convert the value token to an integer in these cases.

9. Last, but not least, we will add the "malfunction" command to our system. Notice that in definitions.h, we have changed the definition of the number of commands to 8 and added a definition for MALFUNCTION to be 54. Now, your job:
   - In the **malfunction** member function of the malfunctionRecord class, if the input time is greater than 0, set the global variable checkEngine to true.
   - In system_utilities.cpp in the **fillCommandList** function add one more set of statements to create a COMMAND object with string "malfunction" and integer MALFUNCTION.
   - In main.cpp, add and implement a MALFUNCTION case of the switch. A malfunction command in the input file will have two additional tokens – the name of the device and the malfunction type. Find the device in the device array and then have that device record a malfunction of the indicated type at the current time.

Checking your work on the EECS Servers

Before turning in your project, its critical that you make sure that your code compiles and runs on the EECS servers. Here's how:

1) Copy your code (all code files + the provided makefile + the input text file) to the eecs server using scp or pscp
2) Connect to the servers (ssh or putty) and in the same directory as the copied files, type 'make' to build your code.
3) Type ./Assignment8 to run your code. Verify that the output is as you expect (using diffchecker.com).

What To Turn In

- Create a **submission.zip** file that contains all the header files and C++ files (i.e. definitions.h, devices.cpp, devices.h, events.cpp, events.h, main.cpp, system_utilities.cpp, system_utilities.h).
- Submit the submission.zip to Canvas. (Do NOT include any of the files generated by your C++ system.)
- Please KEEP all the file names unchanged.

Reminder – be SURE to use diffchecker.com to compare your output to the provided output file. They should be identical (both when run on your machine and on the eecs servers).

Comments

You may get a warning at the end of main.cpp that "delete called on 'device' that is abstract but has non-virtual destructor". This is fine and to be expected. You might also get a bad instruction error (like right before the process ended with exit code message). If the "That's all folks." message has been printed out, and your output up to that point is correct, you don't need to worry about this error.

This assignment is tricky to understand the flow of how everything will happen. Here would be the order of what happens if the human in the car pressed on the break:

1. In the input file, the line "set_device_value brake <value>" appears.
2. That line is read, and we jump to the set_device_value case of our switch in main().
3. We search through the array of devices and find the one with the name "brake".
4. We set the value of the brake device to be whatever the input file specified.
5. Since the global time is greater than 0, we create a new event with the name "brake" and the value specified by the input file.
6. That event then gets added to the linked list of all events, waiting to be processed.
7. We keep reading in lines from the input file until we hit the "time_click" command.
8. At that point, we jump to the time_click case of the switch statement and call **processEvents**().
9. In processEvents, we run through the linked list, processing all events whose process time is at or before the current time.
10. We will process the event created for the brake because the process time of the brake event was set to be the current time, one time click ago.
11. When we process the brake event, we jump to **processBrake()**.
12. In **processBrake()**, we find the brake lamps device and set its value to match the value of the original input from the text file.
13. Since the brake lamps device is a digital controller, we only set the value and do not create a new device.
14. To turn the brake lamp off, we would need to have another line in the input file setting the value of brake to 0.

For something like the turn signals, the process is fairly similar. We would still set the value of the turn signal through the input file and that would change the value of the turn lamp through the intermediate events. In the processTurnLamp functions, we create new events at two time clicks in the future. These new events will flip the value of the turn lamp, essentially making it flash on and off every 2 time clicks. When the human turns the turn signal off in the actual car, a line in the input file would appear to set the value of the turn signal device to off. When this happens, we remove all the events named turn lamp from our linked list, essentially stopping the turn lamp from flashing and shutting it off.