Event-based Programming

Instructor: Numair Khan

Introduction

For our very first lab, we will be implementing a very simple event-driven programming framework using function pointers in C. As the name suggests, in the event-driven programming model the flow of the program is controlled by *events*. What defines an event is determined by your program; for a graphical user interface, actions like mouse-clicks are events. For a browser, the arrival of data packets is an event. Exceptions thrown for error conditions may also be considered events. Events may also be defined at a higher level of abstraction. In computer games, milestones like collecting a certain number of gold coins, or entering a certain level may be defined as events that trigger certain responses. In any case, you will have to explicitly define the type of events that can occur in your program.

You have probably already written very simple event driven programs like the one shown below

```
int main() {
  char c = '\0';
  while(c != 'q') {
    fprintf(stdout, "Enter any key. Press 'q' to quit.\n");
    c = getchar();
  }
}
```

In this simple example, the program waits for an event (a key press), and executes some action in response. Notice, however, that in case you wanted to add a different event (print a greeting when the user presses the 'k' key, for example) you will have to change a significant portion of your code and recompile the program. For a large number of events, the code can become quite unwieldy. So our goal for this lab will be to design our event-driven programs such that they are flexible, modular, and easily extensible.

Two important components of any event-driven program are **the main loop**, and **event handlers**. The main loop listens for any event to occur, and then notifies the event

handlers (also called *callback functions*) which take any necessary action. Event handlers are usually written by users of your code/framework. If you are familiar with web development in javascript, you have probably written functions to handle events like button clicks, form submissions, etc. These functions, which you - the end programmer - write, are the event handlers. The main loop on the other hand is invisible to you in javascript. The main loop in event-driven programming is written by the original author of the code.

Task

to implement a basic event-driven framework with two events:

1. `onProgramStart` - event occurs when the program starts
2. `onKeyPress` - occurs when any key is pressed

define an `enum` type with an id for each of these events; or alternately, you can use the `#define` directive:

```
#define eventid_onProgramStart 0
#define eventid_onKeyPress 1
```

The user of your framework should have the option of registering event handlers with your framework. A function with the following signature should be used for this:

```
void addEventListener(int eventId, EventHandler eh);
```

Where `EventHandler` is a function pointer type that you define. The user can also remove existing handlers:

```
void removeEventListener(int eventid, Eventhandler eh);
```

One possible usage scenario is illustrated below:

```
void printWelcomeMessage() {
   printf("Welcome to my event-driven program.\n");
}

int main() {
   ...
   addEventListener(eventid_onProgramStart, printWelcomeMessage);
   ..
}
```

In this example, the `printWelcomeMessage()` function should be called when the program starts.

Your framework should not impose any limits on the number of event handlers that can be registered for a single event. As discussed in class, the way to proceed would be to define a dynamic array of function pointers for each event type. When the user registers a function as an event handler, add it to the array of the corresponding event. Then, when that event occurs, iterate through the array and call all registered event handlers through their stored function pointers.

Also recall that function pointers can only point to functions with particular signatures (that is, arguments and return types). So what should be the signature of the event handlers that can be legally registered with your framework? This depends on the kind of information that needs to be passed to each handler. In our case, the event that occurs when the program starts does not need to provide the event handlers with any additional information. However, for the key press event, the handler may want to know which key was pressed. This information should be passed as an argument to the event handler, and your function pointer type should reflect this signature. A good way to proceed is to have your function pointer type accept as an argument a void pointer. This can be set to NULL if no data is to be sent to the handler. So, the following function CAN be registered as an event handler:

```
void legalEventHandler(void *data) {
   if(data == NULL) { } // no data
   else { //process data through the void pointer }
}
```

But the following function CANNOT be registered as an event handler:

```
void illegalEventHandler() {
   //legal event handlers need to accept a void* as an argument
}
```

Delivarables

1. Your framework should provide the functionality to add multiple event handlers for each event type through the `addEventListener()` function.
2. Your framework should allow an existing handler for an event type to be removed through the `removeEventListener()` function.
3. Your framework should call all registered event handlers when any of the three events occurs.

Your code will be tested by registering event handlers and checking that they are called at the appropriate time. The event handlers will be registered in a function called `init()`.

You must call this function from your `main()` function at a point where your framework has been initialized.

.