COMP 161 - Lecture Notes - 10 - State and Functions

Spring 2016

We now step back and consider a complete program from start to finish. Along the way we'll see examples of problems that are succinctly captured with state and the use of functions vs mutators.

The Program

The program we'll consider is a simple interactive "game" that functions off a REPL interface. The game begins with the player on the first of 21 spots. Players choose some integer number of places to move. Their piece is then moved that many places. If while moving they go past the first or last spot, then their piece wraps around to the other side. The game tracks the number of times they wrap around. That's it.

The interface for the game should show their piece as an X on a line as well as display their wrapped score. The use is then prompted for their move. The game then updates their location and score and the loop repeats. Figure 1 shows what that would look like for a short game.

A Problem of State

This problem clearly involves state. At any given time we must know two things: the player's location and the number of times they've wrapped around. It makes sense to look at these these things as values that change *over time*. Anytime you fix your logic on how a piece of information changes over time, then you're looking at a situation where state is an obvious choice. The information is represented by a variable and the change is carried out through mutation. Our experience with mutation and state thus far as been largely confined to uses state to solve problems. Now we see that some problems are naturally expressed in terms of state.

At this point can start stubbing out a bit of *main* to capture what we know about our program as C++. In doing so we transliterate high-level, abstract information and design to concrete code. So what do we know? We know that the game operates with a basic REPL design and as it loops it works with two state variables. In figure 2 we see these ideas as C++.

```
|X--------------------|
wrapped: 0

move? 3

|---X-----------------|
wrapped: 0

move? -7

|----------------X----|
wrapped: 1

move? 15

|----------X----------|
wrapped: 2

move?  50

|------------------X--|
wrapped: 4

move ?
```

Figure 1: A Short, Four move game

```
int main( int argc, char* argv[] ){

  // Program State Variables
  int cur_loc{0}; // player location
  int num_wrap{0}; // number of times player wrapped

  while( true ){

    // ... cur_loc ...
    // ... num_wrap ...
  }

  return 0;
}
```

Figure 2: A quick, initial sketch for *main*

Wish Lists and Top-Down Design

Now that we have a very basic starting place, we can begin the process of generating a wish list of procedures that can be used to complete the problem. Notice we do not start completing the program. This procedural design so we start by finding procedures. We also want to give ourselves some options for each part of the program. We'll call this *exploring the program design space*. The first procedure we think of might work just fine, but there are bound to be options and we should consider how those options compare to our first idea.

To find procedures we'll start by thinking big and work our way down to details. Everything that happens in this program clearly happens inside the loop. So we need a sequence of procedures that carry out the different steps of the loop. This is what is typically meant by *top-down* thinking. At the top is the big picture. At the bottom is the micro-level view. For procedural programs in C++ that can mean *main* is at the top and all its helpers[1] are at the bottom. For top-down design, our goal is to write main first, then implement the library needed to complete the main we've written.

[1] out program libraries

The other thing we should keep in mind is that the design of this program already revolves around two state variables. In theory, every procedure we need interacts with these variables in some way shape or form. When we're looking for potential helpers for main, then we can always look for a function, mutator, input, or output procedure that works with one or both of our state variables.

So what happens inside the loop? We can break this game down to three steps:

1. Display the Game State to the user
2. Get the next move from the user
3. Update the game state

Hey! Those could each be procedures. The first is an output procedure and the second is an input procedure. The third step is neither input nor output. The most natural expression of this step is as a mutator that (potentially) modifies both state variables. This is natural because we're thinking in terms of state and the fundamental operation of state is mutation. The word "update" itself implies a $+=$ like operation.

We could rethink the update as assignment ($=$) plus a function. This is *exactly* how you operated in COMP160. A function is used to compute the new value for the state and basic assignment is then used to update the state. We will definitely look at this option when we implement step three, but choosing this option now means we

need to break step three into two steps: update the location and update the wrapped score. Why? Functions can only return a single value and we need two values. Using a mutator we can write a procedure that takes two reference parameters and modifies them both. Alternatively, we could figure out how to use STRUCTS in C++. This would allow us to create a game state struct type that ENCAPSULATED both the location and the wrapped score. The update function would then take one of these structs by value and return one by value. Again, this is *exactly* how you operated in COMP160. Rather than add C++ structs to the mix, we'll work with basic atomic variables[2] and write a double-mutator.

[2] one value. as opposed to compound (struct) data with multiple contained values

Let's go ahead and declare and document these procedures to transliterate our ideas to C++. We'll use several namespaces to organize things. All the procedures will get put in a *movegame* namespace which will act as the programs main namespace. We'll then stick our I/O procedures in a *ui* namespace which is where we'll put procedures that are clearly about the User Interface. Finally, we'll put the update procedure in a *model* namespace as it's all about interacting with our COMPUTATIONAL MODEL of the game state, i.e. those two variables. We'll just go ahead and stick all of this in a single library *move_lib.h*. We could split *ui* and *model* into two libraries, but this program is simple enough that there isn't a good reason to do so. The beginning of our library header is given in figure 3.

In declaring these steps as procedures we're forced to work out some program level details. For starters, we need some local state (*move*) to manage user-input. Otherwise, we need to carefully consider what information each step is dependent upon. Displaying the state requires, well, all the state. Getting the new move requires that local state, but that's it. Finally, updating the state requires the state and the move, but we only need the move value, not the state itself.

We can now actually finish main! This sounds crazy but if we implement those procedures to do what we want them to do, then main should do what it needs to do. Of course *until* the procedures are fully implemented, *main* won't work as intended. That shouldn't stop you. Not at all. Finishing up *main* like this gives you a concrete testable specification to work towards. This is a much better place to be than working towards some undetermined end point. Out procedures should all come together just like we see in figure 4.

At this point we can just stub out the procedures and compile and run our program. It will do nothing, but now we have a complete design for main that we can work towards. Stubs for the top-level helpers can be found in figure 5.

Figure 3: The top-level helpers for *main*

```
// in move_lib.h
namespace movegame{

  namespace ui{

    /**
     * Write the board and number of wraps to the stream out
     * @param loc the location of the player
     * @param wrap the number of times wrapped
     * @return none
     * @pre 0<=loc<21 , 0<=wrap
     * @post representation of the game state is written to the
     *  stream out
     */
    void displayState(std::ostream& out,
              int loc, int wrap);

    /**
     * Get the number of spaces to move from the player
     * @param in the stream where user input can be found
     * @param move the variable where the user's move is stored
     * @return none
     * @pre none
     * @post the user's move (int number of steps) is read from
     *   in
     */
    void getMove(std::istream& in, int& move);


  } // end ui

  namespace model{

    /**
     * Modify the location state and wrapped score based on the
     *  most recent move.
     * @param curr_loc current player location
     * @param num_wrap number of times player has wrapped
     * @return none
     * @pre 0<= cur_loc < 21. 0= num_wrap.
     * @post curr_loc moved move spaces and num_wrap is
     *  incremented accordingly
     */
    void updateState(int& cur_loc, int& num_wrap, int move);
  }

} //end move
```

```
int main( int argc, char* argv[] ){

  // Program State Variables
  int cur_loc{0}; // player location
  int num_wrap{0}; // number of times player wrapped

  while( true ){
    // write out game state
    movegame::ui::displayState(std::cout,cur_loc,num_wrap);

    // get the next move
    int move{0}; // user's move
    movegame::ui::getMove(std::cin,move);

    // update the state
    movegame::model::updateState(cur_loc,num_wrap,move);
  }

  return 0;
}
```

Figure 4: The complete definition of *main*

```
// in move_lib.cpp

void movegame::ui::displayState(std::ostream& out,
                        int loc, int wrap){
  return;
}

void movegame::ui::getMove(std::istream& in, int& move){
  return;
}

void movegame::model::updateState(int& cur_loc, int& num_wrap, int
    move){
  return;
}
```

Figure 5: Top-level procedure stubs

Getting the User's next Move

Display the Game State

Updating the Game State