COMP 161 - Lecture Notes - 10 - State and Functions: Putting It All Together

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 we can use to complete the problem. Notice we do not start completing the program by writing statements. This procedural design so we start by finding procedures. *Your goal is to thing through the problem as procedures*. Why? Procedures are flexible because they are abstract. You can build any procedure you can imagine. Statements are fixed and constrained by the operations and procedures that already exist in the language and libraries. Resist the urge to write statements and instead imagine procedures.

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 third 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 the assignment ($=$) of the return value of 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

At this point we might work out our ideas by actually completing main with procedures. The idea is to work out the details of the declaration and documentation of a procedure by working it in context. The power of this technique shouldn't be under estimated. Just like writing tests for procedures prior to implementing procedures lets you think through the expected behavior of the procedure, using procedures in main[3] lets you think through the purpose and signature of a procedure before "officially" documenting and declaring a procedure. It's all about establishing the *what* of your program specification and implementation before worrying about the *how*.

[3] or where ever you might call them

We'll plan to 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.

If we stick to basic procedure design[4], then we're likely to end up with something like what we see in figure 3. On the other hand, it might be nice to combine the user input prompt with the user input itself. This requires a hybrid I/O procedure. We haven't done that but it's not a big stretch. The basic procedure types aren't the only possible options, they're just the fundamental building blocks. *They are are bottom*. When we combine them into multipurpose procedures, we should be ready to jump right to more primitive helpers. If we allow ourselves some hybrid-purpose procedures then we end up with what we see in figure 4.

[4] A procedure is either a function, a mutator, an input, or an output procedure

Let's go ahead and declare and document these procedures to

```
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);

    std::cout << '\n';
    std::cout << "move? : ";
    // get the next move
    int move{0}; // user's move
    movegame::ui::getMove(std::cin,move);

    std::cout << '\n';

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

  return 0;
}
```

Figure 3: The complete definition of *main*

```
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);

    std::cout << '\n';

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

    std::cout << '\n';

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

  return 0;
}
```

Figure 4: The complete definition of *main* ver. 2

transliterate our ideas to C++. The beginning of our library header is given in figures 5 and 6.

Figure 5: The top-level ui 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);

    /**
     * Prompt the user for the number of spaces to move and get
         that
     * number from the player
     * @param out the stream where the prompt is written
     * @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 a prompt is written to out and the user's move
     * (int number of steps) is read from in
     */
    void getMoveWithPrompt(std::ostream& out, std::istream& in,
        int& move);


  } // end ui
} //end movegame
```

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

Figure 6: The top-level model helpers for *main*

In declaring these steps as procedures and working them in their desired context 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 and if we want to prompt with the input we need an ostream. Finally, updating the state requires the state and the move, but we only need the move value, not the state itself. We also can think about spacing the different output. Should the procedures pad spacing like our example or should we manage that within *main* itself.

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 7.

Display the Game State

Before we do anything, let's write tests for *displayState*. We shouldn't consider how we'll implement this thing until we're certain what it should do. There aren't really any cases to *displayyState* procedure assuming that all the preconditions are met[5]. For the sake of our understanding, figure 8 provides a few different test cases.

[5] which we can manage through updateState

First things first, we need to recognize that the game's state, taken as abstract information, is compound. It's the combination of the lo-

```
// 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::ui::getMoveWithPrompt(std::ostream& out,
    std::istream& in,
                       int& move){
  return;
}

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

Figure 7: Top-level procedure stubs

cation and the wrapped score. While our physical representation of it is as two atomic variables, the logic of our design should mirror the reality of the problem whenever possible. We learned from COMP160 that when faced with compound data, we should deconstruct the pieces and use helpers. In this case that means two auxiliary procedures, an output procedure for each state variable.

We know we need two procedures, but what should they do and how will we use them? At this point we have two ways to proceed: functionally or statefully. The later approach means each helper is an output procedure and the design goal is two simply decompose the compound output into two distinct output procedures. We'll call them *displayLocOnBoard* and *displayWrap* and they output the board and the wrapped score respectively. As seen in figure 9, we can combine them through SEQUENTIAL STATEMENTS or if you want to chain them we could use returned references to do a single statement with NESTED EFFECTS as seen in figure 10. To leave our options open we can simply implement them with returned references and then choose which style we prefer[6] In both cases, we might give serious consider to the creative use of I/O manipulators *std::setw*, *std::setfill*, and possibly the alignment manipulators to solve this problem quickly and easily while keeping the logic squarely in the realm of I/O.

[6] You can always ignore the returned reference because the return value isn't necessary for producing the desired effect.

The functional option is less obvious because we solve an output

Figure 8: Tests for *displayState*

```
 TEST(dispSt,all){
   std::string expected{""};
   std::ostringstream actual{""};

   expected = std::string("|X--------------------|\n");
   expected += std::string("wrapped: 0\n");
   movegame::ui::displayState(actual,0,0);

   EXPECT_EQ(expected,actual.str());

   actual.str("");
   actual.clear();
   expected.clear();

   expected = std::string("|-----X---------------|\n");
   expected += std::string("wrapped: 4\n");
   movegame::ui::displayState(actual,5,4);
   EXPECT_EQ(expected,actual.str());

   actual.str("");
   actual.clear();
   expected.clear();

   expected = std::string("|--------------------X|\n");
   expected += std::string("wrapped: 2\n");
   movegame::ui::displayState(actual,20,2);
   EXPECT_EQ(expected,actual.str());

   actual.str("");
   actual.clear();
   expected.clear();

 }
```

Figure 9: *displayState*

```
// in move_lib.cpp

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

  // Sequential Statements called for effect
  movegame::ui::displayLocOnBoard( out , loc );
  movegame::ui::displayWrap( out , wrap );
  return;
}
```

Figure 10: *displayState*

```
// in move_lib.cpp

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

  // Nested Output effects, i.e. effect chaining
  movegame::ui::displayWrap(
    movegame::ui::displayLocOnBoard( out , loc ),
    wrap) ;
  return;
}
```

problem by first doing something other than output. It is appealing though because the it more clearly separates UI code from model code. This underlying design goal is known as SEPARATION OF CONCERNS. The output procedure, which is part of the UI, doesn't need to know anything about the game state because the model procedure will be designed to provide an appropriate string for the UI to output. The role of *displayState* is now simply to invoke the model procedures that produce the two strings it needs to output and then output those strings. We take this design for a spin in figure 11.

Figure 11: *displayState*

```
// in move_lib.cpp

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

  std::string boardStr{ movegame::model::boardString(loc) };
  std::string wrapStr{ movegame::model::wrapString(wrap) };

  out << boardStr << wrapStr;

  return;
}
```

We have two design options for *displayState*'s helper procedures. The first simply decomposes the output task to mirror the logical structure of the state it's meant to output. We can finish that design up using either sequential statements or chained effects in a single statement and the helpers *displayLocOnBoard* and *displayWrap* can be designed and implemented to support either option. Our second design uses the functions *boardString* and *wrapString* of the state variables *loc* and *wrap* respectively to produce strings that are then output by *displayState*. This design illustrates an important technique in

program design. When programs are software that is developed and maintained over time, then the separation afforded by this design eases the burden of typical software maintenance tasks[7]. In lab you'll complete the design and implementation of all four of the functions so that you can then choose one of the *displayState* implementations above.

[7] fixing bugs and adding features

Updating the Game State

Before we do anything, let's write tests for *updateState*. We shouldn't consider how we'll implement this thing until we're certain what it should do. In writing tests we need to carefully analyze the cases of this problem. This problem is compounded by the fact that we're dealing with a compound state. Naively, this means we might do a case analysis of each individual state variable. For the location, there appears to be three cases: the move doesn't wrap around, it wraps around the left hand side, it wraps around the right hand side. Wrapping an happen multiple times so we should first test a single wrap then test more than one wrap. A quick analysis reveals that these are in fact the same cases we find for the warp count state. All told, we have 5 cases and we need to test the effect of each case on both state variables.

Now that we have a clearer picture of what *updateState* should do we can turn our attention to the implementation design. Once again we're presented with two familiar options: decompose the mutator into two single variable mutators or design two functions that compute the next value for *updateState* to assign to the variables. In both cases we find that computing the next wrapped value requires the value of the current location[8]. This causes a bit of a problem as we need to be certain that we use the original location to compute the new wrapped score. We see the mutator based solution in figure 13 and the functional solution in figure 14.

[8] but not the state itself!

The two implementations of *updateState* shown here bear careful consideration as their equivalency is a simple demonstration of how one can use either functions or state mutation to achieve a computational solution to a problem. For lab you'll implement all four of the helpers listed above.

Getting the User's next Move

Before we do anything, let's write tests for *getMove* and *getMoveWithPrompt*. We shouldn't consider how we'll implement this thing until we're certain what it should do. If we assume valid user inputs[9], then there's not much in the way of cases here either. As seen in figure

[9] a poor assumption but we'll make it for now

```
TEST(udtSt,all){
  int loc{0};
  int wrap{0};

  movegame::model::updateState(loc,wrap,3);
  EXPECT_EQ(3,loc);
  EXPECT_EQ(0,wrap);

  loc = 0;
  wrap = 0;
  movegame::model::updateState(loc,wrap,-2);
  EXPECT_EQ(19,loc);
  EXPECT_EQ(1,wrap);

  loc = 0;
  wrap = 0;
  movegame::model::updateState(loc,wrap,25);
  EXPECT_EQ(4,loc);
  EXPECT_EQ(1,wrap);

  loc = 0;
  wrap = 0;

  movegame::model::updateState(loc,wrap,50);
  EXPECT_EQ(8,loc);
  EXPECT_EQ(2,wrap);

  loc = 0;
  wrap = 0;
  movegame::model::updateState(loc,wrap,-50);
  EXPECT_EQ(13,loc);
  EXPECT_EQ(3,wrap);
}
```

Figure 12: Tests for *updateState*

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

  movegame::model::updateWrap(num_wrap, cur_loc, move);
  movegame::model::updateLoc(cur_loc , move);

  return;
}
```

Figure 13: *updateState* done with two mutators

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

  num_wrap = movegame::model::nextWrap(num_wrap, cur_loc, move);
  cur_loc = movegame::model::nextLoc(cur_loc , move);

  return;
}
```

Figure 14: *updateState* done with two functions

15, the tests for the basic *getMove* is standard input testing. On the other hand, the test for *getMoveWithPrompt* requires us to test the compound effects of input and output. We see this in figure 16.

```
TEST(getmv,all){
  std::istringstream in{""};
  int mv{0};

  in.clear();
  in.str("5");
  EXPECT_EQ(0,mv);
  movegame::ui::getMove(in,mv);
  EXPECT_EQ(5,mv);

}
```

Figure 15: Tests for *getMove*

```
TEST(getmvprompt,all){
  std::istringstream in{""};
  std::ostringstream out{""};
  int mv{0};
  std::string expected{""};

  in.str("5");
  expected = "move? ";
  movegame::ui::getMoveWithPrompt(out,in,mv);
  EXPECT_EQ(5,mv); // the input effect
  EXPECT_EQ(expected,out.str()); // the output effect

}
```

Figure 16: Tests for *getMove*

If we step back from *getMoveWithPrompt* for a second, then we can see an obvious decomposition of the hybrid I/O task into the output and input task as seen in figure 17. The input task can be

accomplished with *getMove* so all we need is an output procedure to display the prompt. Let's call it *movePrompt*.

Figure 17: *getMoveWithPrompt* Basic design

```
void movegame::ui::getMoveWithPrompt(std::ostream& out,
    std::istream& in,
                     int& move){
  movegame::ui::movePrompt(out);
  movegame::ui::getMove(in,move);

  return;
}
```

This procedure is so simple that we might just make it a statement. Then again, it's so simple that doing it as a procedure wouldn't take much time. The whole thing is done in figure 18.

Figure 18: *movePrompt*

```
// in the library header (within movegame::ui

   /**
    * Display the getMove prompt on the stream out
    * @param out the stream where the prompt is written
    * @return none
    * @pre none
    * @post prompt written to out
    */
   void movePrompt(std::ostream& out);

// in the tests

  TEST(movePrompt,all){
    std::string expected{"move? "};
    std::ostringstream actual{""};

     movegame::ui::movePrompt(actual);
     EXPECT_EQ(expected,actual.str());
  }

// in the implementation

  void movegame::ui::movePrompt(std::ostream& out){

    out << "move? ";
    return;

  }
```

Big Picture

Two very important design principles came up in our work with this program.

1. Decomposition of a task/procedure into smaller, more constrained tasks/procedures of the original task/procedure and recombine the results.
2. Draw a clear separation between UI and Model tasks using procedures on model state to provide necessary elements of the UI and using UI procedures to manage basic I/O tasks only.

These principles are not disjoint and can work in tandem to complete the design for a program. In lab you should play around with all the variations of this program we discussed in these notes and in class and then consider other procedural designs that could be used to complete this program.

The other thing we looked at is our ability to use both functional thinking and effectful thinking to solve problems. Our core model update procedure can be implemented using either mutators or functions. The same is true of *displayState*. What that means is you can often use functions to implement an effect. We've seen in previous assignments that local mutation of a pass-by-value parameter can be used to implement a function[10]. Putting these two together means that functions can implement effects and effects can implement functions. As the designer and implementer of a program, you can choose which suits your needs and requirements best.

[10] see the shorten function from lab a few weeks ago