

# *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 user 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 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 has 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 think 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<sup>1</sup> 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.

<sup>1</sup> 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<sup>2</sup> and write a double-mutator.

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<sup>3</sup> 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*.

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<sup>4</sup>, 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.

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

<sup>2</sup> one value. as opposed to compound (struct) data with multiple contained values

<sup>3</sup> or where ever you might call them

<sup>4</sup> A procedure is either a function, a mutator, an input, or an output procedure

---

```
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 figure 5.

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 6.

### *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 *displayState* procedure assuming that all the preconditions are met<sup>5</sup>. For the sake of our understanding, figure 7 provides a few different test cases.

<sup>5</sup> which we can manage through *updateState*

### *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<sup>6</sup>, then there's not much in the way of cases here either. As seen in figure 8, 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 9.

<sup>6</sup> a poor assumption but we'll make it for now

### *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 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

```

Figure 5: The top-level helpers for *main*

```

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);

```



---

```
// 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 6: Top-level procedure stubs

Figure 7: 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 8: Tests for *getMove*


---

```

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);
}

```

---

---

```
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 9: Tests for *getMove*Figure 10: Tests for *updateState*