COMP 161
Lecture Notes 12
UI Loops

March 26, 2017

In these notes we look at some basic user interface loop patterns.

Loops at the User Interface Level

The only loops we've looked at so far arose from the need to step through or TRAVERSE the characters in a string[1]. Several situations occur in interactive programs that require the repetition of user interactions:

[1] or more generally the elements of a collection

- The program repeats a single process until the user hard-quits the program. This requires an infinite loop.
- An alternative to forcing a hard-quit from the program is prompt the user after each execution of the core task to see if they'd like to quit. Another form of this is to offer a menu of choices with a quit option. In both cases, we're effectively replacing an infinite loop with a loop that terminates on a specific user input. We'll call this the until-quit loop as it will repeat until the users chooses to quit.
- The program requires a specific input from the user, like a number from one to ten, and so it gets a single input, checks or validates it, and if it doesn't meet the criterion, then it reports an error and restarts the process. This is what we'll call a validation loop.

We'll explore the design of each of these loops below.

Infinite Loop

As we've already seen, a simple infinite loops provides the means to developing a basic REPL[2] interface. Such an interface lets the user repeat a task over and over. Either the while loop or the do while loop can be used as seen in Figure 1 and Figure 2.

[2] Read, Evaluate, Print, and Loop

Figure 1: An infinite *while* loop

```
while(true){
  //repeated process here
}
```

We've mostly dealt with simple, state-less tasks in which each repetition is independent of the last. There is no requirement that the

```
do{
  //repeated process here
}while(true);
```

Figure 2: An infinite *do while* loop

repeated process be state-less however. The simple game from lecture notes 10 used an infinite loop to manage and modify the state of the game. This allowed us to develop a system in which we continually interact with and modify the program state from one iteration of the loop to the next.

The downside of the infinite loop is the fact that it never ends. The only way to terminate such a process is from without. In a Linux command-line environment we use Ctrl-C to kill the currently executing program and return to the shell. That is, we use interrupts at the operating system level to halt the execution of our program. This is usually not ideal. Imagine you wished to save the game state in order to pick up where you left off next time you run the game. Killing the programming does not allow for this. Instead, we need the program to manage its own termination.

Until-Quit Loop

A simple addition to the infinite loop is logic necessary to allow the user to terminate the program directly. This usually means adding a simple I/O sequence asking if they'd like to end the program or conversely if they'd like to quit.

If at least one user interaction is required to terminate the loop and thereby the program, then a do while loop makes perfect sense as it is designed to repeat one more more times[3]. In Figure 3 we see an example of a basic do while loop for user directed termination. A boolean variable is used to track the state of the loop. If it's true, then the loop should continue, if not, the loop terminates. We'll call these kinds of booleans flags.

[3] the for and while loops do zero or more repetitions

Notice that with the addition of the continuation flag, this loop takes on the same fundamental, iterative structure as our counting loops. There is a state variable, continuation, the loop carries out some process, updates the state variable, and terminates when the state variable reaches a particular state. In this case, the state is boolean and updated at the whim of the user. The loop will terminate if and and only if the user enters something that the computer reads as the y character. Counted loops, when written properly, can typically be analyzed such that we know exactly how many repetitions will occur prior to termination.

```
// This boolean acts as a flag for the state of the loop
bool continue{true};
do{

 // Do Core Repeatable Task

 // Prompt for Continuation
 char uin{Y};
 std::cout << "Continue?(Y or N) ";
 std::cin >> uin;
 // Update Continuation Flag
 continue = (uin == 'Y') || (uin == 'y');

}while( continue );
```

Figure 3: Example of an Until-Quit *do while* loop

Input Validation

Imagine you're writing a menu-driven program. Users must select one of 10 menu options by entering a number between one and ten. What should we do when they give us something other than one of these numbers? We can terminate the program with an error, but that's a bad idea from a usability standpoint. A better, more humane idea is to report the error to the user and prompt them to try again. This UI process is captured by an input validation loop. This loop will get an input from the user, validate the input against some condition, and repeat if the input is invalid. Figure 4 gives an example of such a loop.

A key distinction between validation loops and the other loops we've explored is that it's not meant to carry out some repeated core computational task, it's used to guarantee that the task can proceed with its preconditions met. We might, we'll often want to put a second loop around the validation loop to repeatedly get and use valid inputs from the user.

Program Design and UI Loops

As always, we should generalize the overall processes discussed here as procedures. In this case, the procedure *abstracts* away the loop and allows us to put that process to use in a larger context. A decent rule of thumb is to always place loops inside their own procedure. We shouldn't follow this without some reasoning behind it though.

Let's consider a toy example.

A user is repeatedly prompted for a number between one and five. The

```
int uin{0}; //initialize state
bool invalid{false}; //initialize validation flag

do{

  std::cout << "Enter a Number from 1 to 10: ";
  std::cin >> uin;
  invalid = ( uin < 1) || (uin > 10);
  if( invalid ){
    std::cout << "Expected a number between 1 and 10 but got "
          << uin << '". Please Try Again.\n";
  }
}while( invalid );

// Do something relative to the value of uin.
```

Figure 4: Example of an Validation *do while* loop

numbers they enter are multiplied together. After each number entry they are prompted to see if they want to quit. When they quit, the final product is printed out.

This problem can clearly makes use of an until-quit style loop to manage repeatedly getting numbers and updating the product. Getting a valid number requires a validation loop. We could just cram all of this in *main*, but we should break it down to basic procedures instead. To do this we must first identify the program's state.

This toy program manages a product of integers as the user adds new numbers to the product. Thus, the state is a single integer. The state of the program provides an anchor to our design. In the *main* procedure we can declare an initialize state. From there we hand it off to a series of procedures. First, we consider the main loop. What does it do? How to we categorize it's behavior[4]? It will create some output, request some input, and modify the program's state. It is therefore a hybrid of all our effects. In Figure 5 we see how we'd put it to use in as a helper to *main*.

[4] Functional, Mutator, Input, or Output?

```
int main(int argc, char* argv[] ){
  int product{1};
  toy::mainloop(std::cout, std::cin, product);
  return 1;
}
```

Figure 5: The *main* procedure for our product program

The procedure *mainloop* carries out the core loop that repeats until the user quits. This task requires two more loops: one to get and

validate the next number and one to get and validate user input for continuing or terminating the loop. These loops can be tucked away into helpers as well. In Figure 6 we see one way of organizing this logic.

Figure 6: The

```
void toy::mainloop(std::ostream& sout, std::istream& sin,
              int& prod_state){

  bool continue{true};
  do{
    // Get the next user input. Validate.
    int nextNum{1};
    toy::getValidNextNum(sout,sin,nextNum);

    // Update the product with the nextNum
    // Could use Mutator! toy::updateProd(prod,nextNum);
    // but it's simple enough...
    prod_state *= nextNum;

    // Get Continuation. Validate.
    char cont{'Y'};
    toy::getValidContVal(sout,sin,cont);
  }while( continue );

  return;
}
```

In Figure 7 we see an implementation of the procedure for getting a valid number from the user. We could break this apart into atomic procedural units, but it's pretty straight forward from here so we'll just get right to it. Notice we hold off on updating the state until we've validated the user's input. This is good, safe programming that guarantees that so long as the state was initially valid it will never take on an invalid value.

By organizing the program design around the core state variables we can begin to design the program around the effects related to that state. This can lead to highly effect-driven designs, but as we saw in lecture notes 10, we can often bend effect-driven problems around to functional problems by designing functional procedures that return new state values and strings for output. Input can be made functional by returning the user input rather than setting the variable directly. Experiment with both options. They both have merits and are worthy tools for your program design tool box.

```
void toy::getValidNextNum(std::ostream& sout, std::istream& sin,
                  int& next){
   bool invalid{false};
   do{
     sout << "Enter a number between 1 and 5: ";
     int uin{0};
     sin >> uin;

     invalid = !(uin >= 1 && uin <= 5);
     if( invalid ){
       sout << "Expected a number between 1 and 5. Got "
           << uin << ". Please Try Again.\n";
     }
     else{
       // Don't update the state until it's validated
       next = uin;
     }
   }while( invalid );
   return;
}
```

Figure 7: The procedure toy::getValidNextNum