COMP 161 - Lecture Notes - 08 - Completing a program

Spring 2014

In these notes we look at writing the *main* procedure and some common control logic for CLI tasks.

The C in MVC

Thus far we've talked about the kind of procedures that we'll need to capture the program logic involved in the MODEL and the VIEW[1]. To bring all of this together, we need the program CONTROL logic. This is the glue that ties together I/O code with core program/problem computations. Let's consider the following program:

[1] or UI

> The *guess* program plays a simple guessing game with the user. It uses an interactive, text-based, command line interface. When the users starts the program, they're prompted to guess a number between 1 and 10. The program should tell them when they provide guesses that are out of bounds and allow to them to guess again. The program should provide feedback as to whether or not their guess is too high or too low. When they guess the right number, the program terminates.

You should be able to win this game in something like three guesses[2]. For the time being our goal is to implement this game in a correct, simple, and efficient manner. The question now is, where to do put this control code? In C++ programs, we always start with a very specific procedure named *main*.

[2] this is a problem for another day

The main procedure

When a program executes, it executes the *main* procedure. The *main* procedure is unique in that an executable program can have only one main and you, the programmer, never explicitly invoke it. When linking, the compiler looks for a *main* procedure, and builds the executable around it. Until now, we've used a pre-written main provided by our testing framework and haven't had to deal with *main*. Now, we'll look at writing a main.

First things first, we'll always stick main in a file by itself. Procedures written in a library can be reused in other programs and linked with tests. The restrictions on *main* mean we cannot reuse anything within *main* nor can we link anything in the same file as *main* with our testing framework's main. In short, *we're going to keep as much*

code out of main[3] *as possible so that we can more easily test it and potentially reuse it in future programs.*

When we set out to write our own *main* procedure, then we put it in a *cpp* file[4]. The signature of main is the same every time[5] so let's just jump right to a stub.

[3] and the main file

[4] with the usual file header comments

[5] you can use other signatures, but they don't differ too much and we're not going to use them

```
int main(int argc, char *argv[]){
  return 0;
}
```

Based on this signature we can see that all C++ programs must return an integer. This is typically an error code or some sort. When the program terminates under normal conditions, you're expected to return 0[6]. Otherwise, you return some integer that encodes some information about how or why the program terminated.

[6] hence using it as a stub

The two arguments to *main* are used to manage command line arguments. Our program doesn't use command line inputs, so for the time being we'll say just a few things about the inputs. The first input is the *argument count*, and is always at least 1 because the system includes the name of the program as an argument. The second input is all of the *argument values* represented as C STRINGS[7] and stored in an ARRAY[8] The first thing in *argv* is the name of the program as typed by the user. An important observation to make is *argc is the number of C strings stored in argv*. That is, if someones types the name of the program, plus 3 inputs, then *argc* is 4 and *argv* contains the name of the program and the three inputs in the exact order the user typed them.

[7] char *

[8] holds data like a list. more on this later.

High-Level Program Design

First we think about two key elements of the program:

1. What's the model and what state do we need to track? (Current Guess & Target Number)
2. Overall program flow and sequence of events

This STATEFUL perspective of our program is new. Think in terms of time. Find state by considering the question: "At any given moment during the course of the game, what do we need to know to continue on to the next turn?" For the guessing game, we have to know what the user has guessed and the target number. To really narrow things down, decide if any of this information can be thrown away from one turn to the next. We don't really need to remember the previous guess, but we still need a state variable for the guess in order to manage user input.

The overall program flow in interactive programs often involves some initialization code and then some repeated steps.

```
1. Declare and initialize state
2. Compute random target value
3. Repeat until the user wins:
   1. Repeat until the user enters a proper guess
      1. get user's guess
      2. If the guess is invalid, provide a helpful error message
   2. Check the (valid) guess and report if it's too low, too high, or correct
```

Logic expressed in terms of *until* conditions is natural, but we'll need to convert it to the logical opposite, *while* or "as long as". This repetition in code is often carried out by LOOPS[9]. In this program we see two common loop *patterns* for interactive programs:

[9] but can also be done with recursive procedures

1. Validation Loops: Repeat the user input sequence until they give you a valid input value
2. Polling Loop: Repeat until a specific state is detected[10]

[10] The name is derived from the fact that you must poll the system state

The validation loops can be viewed as a specific form of a polling loop where we're polling for valid input. In our game, we must use a validation loop to get the user's guess and a polling loop to repeat "turns" until they've won.

Stubbing out main

We can use our high-level design to better stub out main.

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

  // 1. Declare and initialize state
  int target(0);
  int guess(0)

  // 2. Compute random target value

  //3. Repeat until the user wins:

  //    1. Repeat until the user enters a proper guess
  //      1. get user's guess

  //      2. If the guess is invalid, provide a helpful error message
  //    End 3.1
```

```
  //   2. Check the (valid) guess and report if it's too low, too high, or correct
  //End 3

  return 0;
}
```