Lab 6

2 October 2014

This week you'll spend a bunch of time reading and playing with my code, and continuing to learn your way around some of tools that will save you some time and help you work with larger programs. To get started, create your `lab6` directory and copy into it all the files from `/home/shared/162/lab6/`.

Making your life easier, part 1: Makefile

The `compile` command automates a lot, but it can still get a bit tedious, especially when compiling involves multiple files and/or a given project involves building multiple executable files. Now you'll see a couple of tricks to keep them more manageable.

First, type

```
make test_Leapfrog
```

and hit enter. You'll see that it automatically runs `compile` a few times; and if you `ls` the directory, you'll see that instead of `a.out` it has created an executable named `test_Leapfrog`, which you can run by typing

```
./test_Leapfrog
```

and hitting enter. Now, just type

```
make
```

and hit enter—this told it to build *all* the available executables. To see the rules that guided that process, edit the `Makefile` that I provided, and read its contents.

In general, Makefile rules have three parts. First, the "target" file that the rule creates; then, after a colon, the "dependency" files that are used in the making of the target file; and third, on the next line, after a tab, the command to run in order to generate the target. What makes makefiles truly fantastic is the dependency management—if you make a change to (only)

`Leapfrog.cpp`, when you run `make` it will see that it needs to regenerate `Leapfrog.o`, and do so; and then, once `Leapfrog.o` is changed, it sees that it also needs to regenerate all of the executable targets as well (and does so). If you modify the `.h` file, it sees that everything depends on that (directly or indirectly), and so it ends up rebuilding everything from scratch. If it doesn't need to rebuild something, it won't: if you run `make` again, it should say that nothing needs to be done.

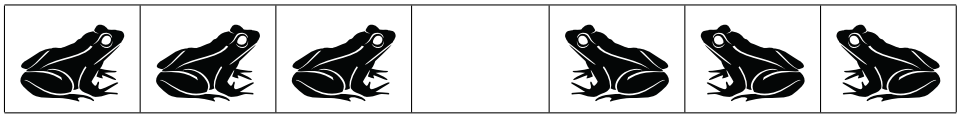
A few other specifics:

- In the executable targets (`test_Leapfrog` etc) you'll notice the symbol combination `$^`. This says, "substitute the entire list of components from the line above". If you need to add a `.o` file to an executable, now you can do it in a single place: the dependency line.
- Further down, there's a series of dependency lists that are specific to each individual `.o` file. These tell the `make` command that if *any* of the other files change, it'll have to recompile the component. Usually a dependency list will include one `.cpp` file and one or more `.h` files.

Exit the editor, and then edit the file `solvefrog.cpp` to add a space or comment or something else that changes the *file* but doesn't affect the program, then save and quit. If you run `make` again, it will detect the "changed" file, but only the targets that depend on that file will be rebuilt.

The leapfrog puzzle

As you may have noticed, the files in this directory have frog-related names; they are all related to a solitaire puzzle that works as follows. There are seven cells filled by frogs facing to the left and right, initially as in this diagram:

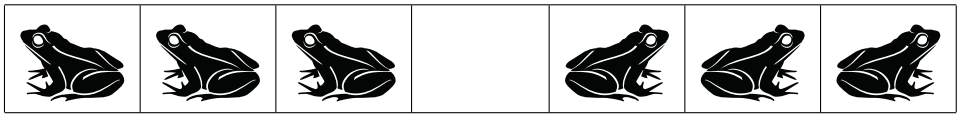


A frog can only be moved into the empty square, and only if the frog is immediately in front of it or can get to it by leaping over a single other frog. For instance, if we choose the third from the end to step forward, the result would be as follows:

and then the frog in front of that one could jump over him:

If you're not careful, you can get yourself stuck:

But if you choose the moves in the correct order, you can make all the frogs swap places before getting stuck:



In this directory, the `Leapfrog` class is designed to simulate the puzzle itself, and the `solvefrog` function (and file) is designed to simulate a player of the puzzle, i.e. someone trying to solve it. Run the different executables that you've built (particularly, `./play_Leapfrog` and `./show_solvefrog`) to understand how the puzzle works. Then, dive in and figure out how the *code* works. Reading someone else's code is an important programming skill.

But simply letting your eyes pass over the code isn't really sufficient when it comes to reading the code; you need to try it out and see what it does. One way to do that is to add print statements (as we've already discussed). The next section introduces a new tool that will let you try some other strategies.

Making your life easier, part 2: debugger

Type `gdb show_solvefrog` to start the debugger on that executable. Once it has started, at the gdb prompt type

```
run
```

in order to run the program. It should run to completion, including output of all the moves to solve the puzzle. You are still inside `gdb` when it completes,

and you can type `run` again; indeed, if you leave gdb open, and in a different window recompile the program, you can type `run` again and it will load in the recompiled program before running it.

Now type (still inside `gdb`)

```
break solvefrog.cpp:34
```

Its response tells you that it has created a "breakpoint" at the specified location; now `run` the program and it will run until it reaches that line, and then it will pause execution and let you inspect the state of things. For instance, you can see the line of code that it's paused on, and that `posn` is a local variable here, so you can

```
print posn
```

and it'll tell you something like "`$1 = 4`"—don't worry about the `$1`, but it's telling you the value of the expression is 4. In fact, if you try it with `retval` you should see something like "`$2 = std::vector of length 1, capacity 1 = {3}`"because at this moment, `retval` is a vector containing only the number 3 (which is the last move required to finish the puzzle). If you look at this area of the code, you'll find that another local variable is `current`, representing the current Leapfrog board state; try printing that too.

Another command you can run to inspect the state of the program is

```
backtrace
```

This shows you the current call stack, closely related to the box trace diagrams we've drawn whenever we talk about recursion; `solvefrog` appears many times because it's a recursive function and we're deep into it. Type `up` and then `print current` again, and you'll see an earlier version of the board. You can use `up` and `down` to navigate the stack and show more of the variable values in it (although, due to a quirky but long-standing convention, up and down go in exactly the opposite direction you'd expect them to). You can at any point type `list` to show you a listing of the part of the program adjacent to the current line.

But there's more! Since the program is still running, you can type

```
next
```

to execute just one statement. Keep pressing Enter and the debugger will repeat the previous command—this lets you go through the program step-by-step and trace *exactly* what statements are being executed.

The three main tracing commands you'll need to use are:

- `next` executes one line of code (the one that the debugger has printed as the next available line to execute).
- `step` is like `next` except that if the line of code includes a function call, it will only execute the first line of that function, and pause again *inside* that function.
- `continue` tells the debugger to resume normal execution and keep running until it hits another breakpoint, or crashes, or the program ends normally.

And don't forget to keep using `print`!

Comprehending the code

Right now, set another breakpoint at line 17, and run the program again. Practice using the three tracing commands, and try to see how the algorithm is executing (it's usually helpful to have the code itself open in another window). My implementation of `solvefrog` is a recursive backtracking algorithm like we discussed in class yesterday, meaning that it recursively tries one thing and, if that turns out not to work, backs up and tries a different thing instead. Note the similarities to other recursive algorithms we've seen, and use the debugger to help you understand what's going on with the call stack.

In the space below, draw a box trace diagram of the call stack, at some point when the stack is four function-calls deep. Remember that each stack frame includes any local variables and parameters. (And definitely use the debugger to help you figure out what you'll need to draw!)

Spend some time digging deeper and looking at the `Leapfrog` class. In particular, the `makeMove` method; it is short because it hands off a lot of work to other code. What is it doing? For each part of `makeMove` that is "handed off" to some other piece of code, write a short description (in English) of what's happening there. (Hint: make use of the method descriptions!)

Look through all the provided files—`.cpp` files, `.h` files, the `.u` files, and the Makefile. In the space below, write down filenames and line numbers of any C++ syntax or usage that you haven't seen before or don't understand. In each case, write down your best guess as to what it does or why it's there. There are a few things I intentionally put in the code that I hadn't shown you yet; part of learning to read code is being able to identify unfamiliar language features, make some educated guesses about them, and then ask someone about them to confirm your understanding.

Handing in

No electronic handin this week; but be sure to bring this lab handout with you to class tomorrow, as we'll be discussing your diagrams, and I'll answer questions/confirm guesses about the unfamiliar C++ features.

Credit: Frog image by Vectorportal.com