Lab 12
Lijnenspel revisited

24 November 2015

Reading the code

The drill this week is to read, analyse, and answer questions about code.

Regardless of how far you got on Lab 8, it will be instructive to read someone else's implementation—mine—and see what's done the same and what's done differently. I've put my implementation in `/home/shared/160-3/lab12`. Copy my implementations into your directory so you can edit them and compile them.

The files `lijnenspel.h`, `lijnenspel.cpp`, and `check_board.cpp` represent a complete implementation of the requirements of Lab 8; `check_board.cpp` has the `main` function and everything else is in the `lijnenspel` files. The file `play_board.cpp` is a further extension designed to support a user "playing" a Lijnenspel board (i.e. trying to solve it).

As you read through the code, you will find it helpful to refer back to the Lab 8 handout. You should also feel free to compile and run the code, and make your own changes to it in order to better understand what each expression or line of code accomplishes. Nothing in any of these files is meant as a particularly fancy C++ trick (although there are one or two things you haven't seen); if you don't understand what something does, you should ask.

To guide your exploration of the code, and also to structure the assignment, I've posed a number of how-and-why questions about my code, which I handed out in class Monday (also available as PDF from the course website). Concise answers are fine (if they're correct!), but make sure they're clear. Include line numbers where relevant. Bring them with you to lab; I'll expect to see an attempted answer for each question. We'll go over the answers early in the lab period before moving on with the lab.

Command line FOTD: `grep`

The `grep` command is a general search tool that lets you find occurrences of some pattern in a whole batch of files. For instance, if you go to the `/home/shared/160-3/lab12` directory (where my implementations are) and type

```
grep grid lijn*.*
```

you'll get a listing of every time the `grid` variable shows up (in either file); what it's doing is going through each source file and printing out every line that contains the string "`grid`".

But `grep` is more powerful than that. Its first argument is what's called a "regular expression" or "regex", and lets you search for some pretty complicated things. You can get more information on this on your own, but a few quick tricks:

- By enclosing the pattern in (single) quotes, you can search for strings with spaces in them:

  ```
  grep 'const Matrix' lijn*.cpp
  ```

- If you want to match any single character, use a period:

  ```
  grep 'int . = 0' lijn*.cpp
  ```

  matches any line that inits a single-letter variable.

- To match any amount of any text, use the period wildcard with an asterisk:

  ```
  grep 'if.*grid' lijn*.cpp
  ```

- To match the beginning or end of the line, use caret and dollar-sign respectively.

  ```
  grep '^int' lijn*.cpp
  ```

  will give just those lines that *start* with `int`.

Vim FOTD: searching

From command mode, if you hit the forward slash key, it's a little bit like colon mode: the cursor moves to the bottom of the screen and awaits further input. But what it's waiting for now is a regular expression to search for.

Having just explained regexes in the context of `grep`, there's not much more to explain here; they work essentially the same way. After the initial slash, you type a regex and hit enter, and Vim will find the next place in the file that matches that regex, or if there are none it will tell you that.

Also inside command mode, the `n` command will repeat the previous search. So pressing `n` repeatedly will cycle through all matches in a file. Using `N` instead goes through matches in reverse order.

The `n` command together with the period command (which repeats the previous command) is a workhorse combination: first, search for a pattern and do something; then alternate `n.n.n.` until you've done your action every place that pattern occurs.

Refining the code

For the rest of the lab (and the next couple of weeks), you'll be refining the Lijnenspel game in various ways. You may either use my code as your starting point, or use your own Lab 8 implementation. If you use your own, be sure to save a copy of the working current version before you start editing it.

Printing the board, part 1

Right now, we're using the `MatrixIO` library to read the grid, which is certainly convenient but not very aesthetic; it prints the board with all these curly brackets and extra space in. For your first "upgrade" to the previous system, you should write a function whose header is

```
void print_board(Matrix<char,2> grid, ostream& out, bool shownumbers);
```

(you can use a const reference to the `Matrix` if you like, but the `ostream` has to have a modifiable reference as I've given it here). It will print the board in a somewhat more compact format: one row per line, but with no

curly brackets and no extra spaces. For instance, this would be the output it gives for a particular partly-filled-in board:

```
...5
.2.v
3..v
.<2>
```

If the boolean `shownumbers` is true, the row and column numbers should be printed as well (to facilitate user input), as shown here:

```
  0123
0 ...5
1 .2.v
2 3..v
3 .<2>
```

Update the code in `play_board.cpp` to call the `print_board` function you've written. Make sure that inside the function you print to `out`, the function's parameter, rather than `cout`—that way your code would work even if it were given an fstream or stringstream.

Verifying user input, part 1

In `play_board.cpp`, introduce some code inside the loop to check the user's input, and if it is input that would not work properly (crashing the program or otherwise breaking something), print a suitable error message and let them input a different move instead.

Reading the board

We're also still using the `MatrixIO` library to read the grid, which is clunky on the input side too. Write a function with the header

```
Matrix<char,2> read_board(istream& in);
```

In the body of the function it can use `in` as a generic name for an input stream (just like `cin` or `ist` or any other input stream we've used). It

should read a number and create a correctly-sized `Matrix` as before; but then it should use `getline` and stringstreams to read the grid and put each character into its appropriate grid position. This should henceforth be a valid input file:

```
4
...5
.2.v
3..v
.<2>
```

Note that you'll have to revise all the input files to match the new format. Update the code in `play_board.cpp` to call the `read_board` function you've written.

Improved data representation

It was always a little grody that the number squares were represented internally as a `char` from `'1'` to `'9'`, requiring us to subtract `'0'` whenever we wanted to make use of them. Let's borrow a trick from the calculator code (from Chapter 6 and Lab 9) and define a `struct` called `Square`: it will have two fields, `kind` and `value`, and when `kind` indicates that the `Square` is a number-square, `value` is the actual integer numeric value stored in that square.

This change will require some substantial code surgery; you may wish to make a copy of your existing code before you implement it.

Once you define the type itself in the `.h` file, you will need to:

- Make every `Matrix` a 2D `Matrix` of `Square` (instead of `char`)
- Edit every access of an element of `grid` to use the named fields—that is, every place there is an access to something like `grid[r][c]` or `grid[i][j]`, that becomes `grid[r][c].kind` and/or `grid[r][c].value` instead.
- That includes the functions `read_board` and `print_board`, which should still handle the same visual representation as before (but when they, for instance, read in a number-square that has a `'4'`, they'll now create a `Square` object whose `kind` indicates it is a number and whose `value` is `4`).

Simplifying user input

Right now, the user input can be rather tedious, since they need to input each cell separately. Modify the user input in `play_board` so that if the user can just specify the *end* of an arrow, along with its direction, and the full length of the arrow is stored. So on the board

```
...6.
5....
..4..
....4
.1...
```

if the user specified row 2, column 3, and a down arrow (`v`), the system would also fill in the cell above:

```
...6.
5..v.
..4v.
....4
.1...
```

and if the user subsequently placed a down arrow at row 4, column 3, the arrow would be extended:

```
...6.
5..v.
..4v.
...v4
.1.v.
```

Printing the board, part 2

Modify your `print_board` implementation so that if an arrow is longer than one cell, it draws it as a single long arrow (using hyphen and vertical bar characters for the middle parts of the arrow). This should *not* cause a change in the internal representation, just in how it's printed.

For instance, the board formerly drawn as

```
^<<6^
5>^v^
v<4v^
v^vv4
v1vvv
```

would now be printed as

```
^<-6^
5>^||
|<4||
|^||4
v1vvv
```

Verifying user input, part 2

Some values of user input wouldn't necessarily *break* anything per se, but would lead to clearly invalid solutions; in particular, if the user tries to draw more arrow leading out of a number-square than that number-square can support, the solution can never even possibly be completed.

Further improve your verification of user input to reject moves that would put a number-square over its quota of outbound arrows. (Politely, and with a chance to keep trying other moves, of course.)

Handing in

As usual, use the `handin` program. Designate this as `lab12`. The final handin for this will be due **at class time on the last day of classes**, 4 December.

Rubric (tentative)

RUBRIC

General

1 Attendance at lab (24 November) with worksheet filled out

1 Present in lab (1 December)

Board I/O

1 `print_board` basically works (v1)

½ ... printing to the provided `ostream&`

½ ... and shows correct row/col numbers, when `shownumbers` is set

1 Input/test files match revised format

1 `read_board` works correctly

1 `print_board` displays long arrows (hyphens, vertical bars)

1 Code in `play_board` calls `read_board` and `print_board`

User interface/input

1 Catches and rejects ≥ 1 kind of invalid input

1 Catches and rejects ≥ 3 kinds of invalid input

1 User input extends arrows ("simplifying user input")

1 Catches and rejects arrows that put number square over quota

`Square` **struct**

1 Define `Square`, create number square and other kinds of value

1 Read and print code work w/ `Square`, store correct number

1 Other lijn functions use `kind` and `value` appropriately