Lab 12 Part 2
Lijnenspel revisited

21 November 2019

Refining the code

For the rest of the lab, you'll be refining the Lijnenspel game in various ways. Your starting point will be my implementation of the first part of Lab 12 with one or two tweaks added. If you want to save your work before copying mine, you can make another directory and copy your own files in, or you can copy today's files into a brand-new directory (such as "`lab12-2`").

Integrating my code

In addition to the `lijnenspel.cpp` file containing the required functions from the first part of the lab, I have written *two* different `.cpp` files with `main` functions: one as a sort of tester system (reads a board from `cin` and prints out some info about it) and the other as a further extension designed to support a user "playing" a Lijnenspel board (i.e. trying to solve it).

Find these files, and a number of test cases designed to work with `check_board`, in `/home/shared/160/lab12/` . Copy them into your directory.

You should take a minute to read and understand the files. Look at `check_board.cpp` first, then `lijnenspel.cpp`, and finally at `play_board.cpp`. Make note of at least one thing that I did differently from how you implemented it, and at least one thing I did in C++ that we haven't seen before, and in about ten minutes we'll talk about some of those things as a group.

The test files are geared towards the current version of the system; as you make changes (below), you may need to make corresponding changes to the test files. Do so!

Printing the board indices

For your first "upgrade" to the previous system, you should edit your printing function to print the row and column numbers as well (to facilitate user

input), as shown here:

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

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.

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 refine our internal representation (the "model") 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. The grid will thus now be a `vector<vector<Square>>` instead of containing `char`. After conversion to this new model, the *internal* representation (model) of a simple partially-solved grid might be

```
{ { {'.', 0}, {'.', 0}, {'#', 3} },
  { {'.', 0}, {'^', 0}, {'.', 0} },
  { {'#', 2}, {'#', 1}, {'.', 0} } }
```

which should still *print* (the "view") as

```
   012
 0 ..3
 1 .^.
 2 21.
```

The difference is that the `kind` field is `'#'` for all number-squares, and the `value` field contains the actual integer value of that number-square (and zero for all other kinds of squares). This adds a little complexity to the I/O functions but substantially simplifies several others.

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 `vector` a 2D `vector` 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]`, 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 ("view") as before (but when they, for instance, read in a number-square that has a `'4'`, they'll now store it as a `Square` object whose `kind` is `'#'` and whose `value` is the integer `4` instead of the character `'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

```
  01234
0 ...6.
1 5....
2 ..4..
3 ....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:

```
  01234
0 ...6.
1 5..v.
2 ..4v.
3 ....4
4 .1...
```

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

```
  01234
0 ...6.
1 5..v.
2 ..4v.
3 ...v4
4 .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 (the "model"), just in how it's printed (the "view").

For instance, the board formerly drawn as

```
  01234
0 ^<<6^
1 5>^v^
2 v<4v^
3 v^vv4
4 v1vvv
```

would now be printed as

```
  01234
0 ^<-6^
1 5>^||
2 |<4||
3 |^||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` (again). The final handin for this will be due **4pm on Wednesday**, 3 December.

Rubric (tentative)

RUBRIC

General

1 Attendance at lab

1 Test cases updated to reflect new output and added to test new features

1 `print_board` shows correct row/col numbers

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

User interface/input

1 Catches and rejects ≥ 2 kind 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