

```
1: /*****
2:  * sudoku.h
3:  *
4:  * Computer Science 50
5:  * Problem Set 4
6:  *
7:  * Compile-time options for the game of Sudoku.
8:  *****/
9:
10: // game's author
11: #define AUTHOR "Doug Lloyd"
12:
13: // game's title
14: #define TITLE "Sudoku"
15:
16: // banner's colors
17: #define FG_BANNER COLOR_CYAN
18: #define BG_BANNER COLOR_BLACK
19:
20: // grid's colors
21: #define FG_GRID COLOR_WHITE
22: #define BG_GRID COLOR_BLACK
23:
24: // border's colors
25: #define FG_BORDER COLOR_WHITE
26: #define BG_BORDER COLOR_RED
27:
28: // logo's colors
29: #define FG_LOGO COLOR_WHITE
30: #define BG_LOGO COLOR_BLACK
31:
32: // built-in numbers' colors
33: #define FG_FIXED COLOR_RED
34: #define BG_FIXED COLOR_BLACK
35:
36: // game won colors
37: #define FG_WON COLOR_GREEN
38: #define BG_WON COLOR_BLACK
39:
40: // nicknames for pairs of colors
41: enum { PAIR_BANNER = 1, PAIR_GRID, PAIR_BORDER, PAIR_LOGO,
42:        PAIR_FIXED, PAIR_WON };
```

```
1:  /*****
2:  * sudoku.c
3:  *
4:  * Computer Science 50
5:  * Problem Set 4
6:  *
7:  * Doug Lloyd
8:  * October 11, 2011
9:  *
10:  * Implements the game of Sudoku.
11:  * EXTRA FEATURES:
12:  *   - cursor wrap-around
13:  *   - all numbers turn green on win
14:  *   - numbers that came with board are red
15:  *   - user can undo their most recent move
16:  *****/
17:
18: #include "sudoku.h"
19:
20: #include <ctype.h>
21: #include <ncurses.h>
22: #include <signal.h>
23: #include <stdbool.h>
24: #include <stdio.h>
25: #include <stdlib.h>
26: #include <string.h>
27: #include <time.h>
28:
29:
30: // macro for processing control characters
31: #define CTRL(x) ((x) & ~0140)
32:
33: // size of each int (in bytes) in *.bin files
34: #define INTSIZE 4
35:
36:
37: // wrapper for our game's globals
38: struct {
39:     // the current level
40:     char *level;
41:
42:     // the game's board
43:     int board[9][9];
44:
45:     // holds whether initial spaces of the board are changeable
46:     bool changeable[9][9];
47:
48:     // the board's number
49:     int number;
50:
51:     // the board's top-left coordinates
52:     int top, left;
53:
54:     // the cursor's current location between (0,0) and (8,8)
55:     int y, x;
56:
57:     // the state of the game as won or not won
58:     bool won;
59:
60:     // a structure that holds information related to the last move
61:     struct {
62:         // the position of the last change
63:         int y, x;
64:     }
```

```
65:         // the value inputted at the last move
66:         int ch;
67:     } lastmove;
68: } g;
69:
70:
71: // provided prototypes
72: void draw_grid(void);
73: void draw_borders(void);
74: void draw_logo(void);
75: void draw_numbers(void);
76: void hide_banner(void);
77: bool load_board(void);
78: void handle_signal(int signum);
79: void log_move(int ch);
80: void redraw_all(void);
81: bool restart_game(void);
82: void show_banner(char *b);
83: void show_cursor(void);
84: void shutdown(void);
85: bool startup(void);
86:
87: // added prototypes
88: void move_cursor(int ch);
89: void insert_symbol(int ch);
90: bool legal_move(int x);
91: bool legal_box(int x);
92: bool legal_col(int x);
93: bool legal_row(int x);
94: bool game_won(void);
95: void undo(void);
96:
97: /*
98:  * Main driver for the game.
99:  */
100:
101: int main(int argc, char *argv[]) {
102:     // define usage
103:     const char *usage = "Usage: sudoku n00b|l33t [#]\n";
104:
105:     // ensure that number of arguments is as expected
106:     if (argc != 2 && argc != 3) {
107:         fprintf(stderr, usage);
108:         return 1;
109:     }
110:
111:     // ensure that level is valid
112:     if (strcmp(argv[1], "debug") == 0)
113:         g.level = "debug";
114:     else if (strcmp(argv[1], "n00b") == 0)
115:         g.level = "n00b";
116:     else if (strcmp(argv[1], "l33t") == 0)
117:         g.level = "l33t";
118:     else {
119:         fprintf(stderr, usage);
120:         return 2;
121:     }
122:
123:     // n00b and l33t levels have 1024 boards; debug level has 9
124:     int max = (strcmp(g.level, "debug") == 0) ? 9 : 1024;
125:
126:     // ensure that #, if provided, is in [1, max]
127:     if (argc == 3) {
128:         // ensure n is integral
```

```
129:     char c;
130:     if (sscanf(argv[2], " %d %c", &g.number, &c) != 1) {
131:         fprintf(stderr, usage);
132:         return 3;
133:     }
134:
135:     // ensure n is in [1, max]
136:     if (g.number < 1 || g.number > max) {
137:         fprintf(stderr, "That board # does not exist!\n");
138:         return 4;
139:     }
140:
141:     // seed PRNG with # so that we get same sequence of boards
142:     srand(g.number);
143: }
144: else {
145:     // seed PRNG with current time so that we get any sequence of boards
146:     srand(time(NULL));
147:
148:     // choose a random n in [1, max]
149:     g.number = rand() % max + 1;
150: }
151:
152: // start up ncurses
153: if (!startup()) {
154:     fprintf(stderr, "Error starting up ncurses!\n");
155:     return 5;
156: }
157:
158: // register handler for SIGWINCH (SIGnal WINdow CHanged)
159: signal(SIGWINCH, (void (*)(int)) handle_signal);
160:
161: // start the first game
162: if (!restart_game()) {
163:     shutdown();
164:     fprintf(stderr, "Could not load board from disk!\n");
165:     return 6;
166: }
167: redraw_all();
168:
169: // let the user play!
170: int ch;
171: do {
172:     // refresh the screen
173:     refresh();
174:
175:     // get user's input
176:     ch = getch();
177:
178:     // capitalize input to simplify cases
179:     ch = toupper(ch);
180:
181:     // process user's input
182:     switch (ch) {
183:
184:         // start a new game
185:         case 'N':
186:             g.number = rand() % max + 1;
187:             if (!restart_game()) {
188:                 shutdown();
189:                 fprintf(stderr, "Could not load board from disk!\n");
190:                 return 6;
191:             }
192:             break;
```

```
193:
194:     // restart current game
195:     case 'R':
196:         if (!restart_game()) {
197:             shutdown();
198:             fprintf(stderr, "Could not load board from disk!\n");
199:             return 6;
200:         }
201:         break;
202:
203:     // let user manually redraw screen with ctrl-L
204:     case CTRL('l'):
205:         redraw_all();
206:         break;
207:
208:     // allow for cursor movement
209:     case KEY_UP: case KEY_DOWN:
210:     case KEY_LEFT: case KEY_RIGHT:
211:         move_cursor(ch);
212:         show_cursor();
213:         break;
214:
215:     // allow for changing of board values
216:     case '1': case '2': case '3': case '4': case '5':
217:     case '6': case '7': case '8': case '9': case '0':
218:     case '.': case KEY_BACKSPACE: case KEY_DC:
219:         insert_symbol(ch);
220:         game_won();
221:         draw_numbers();
222:         show_cursor();
223:         break;
224:
225:     // let the user undo the last thing they did
226:     case 'U': case CTRL('z'):
227:         undo();
228:         draw_numbers();
229:         show_cursor();
230:         break;
231: }
232:
233:     // log input (and board's state) if any was received this iteration
234:     if (ch != ERR)
235:         log_move(ch);
236: } while (ch != 'Q');
237:
238: // shut down ncurses
239: shutdown();
240:
241: // tidy up the screen (using ANSI escape sequences)
242: printf("\033[2J");
243: printf("\033[%d;%dH", 0, 0);
244:
245: // that's all folks
246: printf("\nkthxbai!\n\n");
247: return 0;
248: }
249:
250:
251: /*
252:  * Draw's the game's board.
253:  */
254:
255: void draw_grid(void) {
256:     // get window's dimensions
```

```
257:     int maxy, maxx;
258:     getmaxyx(stdscr, maxy, maxx);
259:
260:     // determine where top-left corner of board belongs
261:     g.top = maxy/2 - 7;
262:     g.left = maxx/2 - 30;
263:
264:     // enable color if possible
265:     if (has_colors())
266:         attron(COLOR_PAIR(PAIR_GRID));
267:
268:     // print grid
269:     for (int i = 0 ; i < 3 ; ++i ) {
270:         mvaddstr(g.top + 0 + 4 * i, g.left, "+-----+-----+-----+");
271:         mvaddstr(g.top + 1 + 4 * i, g.left, "|           |           |           |");
272:         mvaddstr(g.top + 2 + 4 * i, g.left, "|           |           |           |");
273:         mvaddstr(g.top + 3 + 4 * i, g.left, "|           |           |           |");
274:     }
275:     mvaddstr(g.top + 4 * 3, g.left, "+-----+-----+-----+" );
276:
277:     // remind user of level and #
278:     char reminder[maxx+1];
279:     sprintf(reminder, "    playing %s #d", g.level, g.number);
280:     mvaddstr(g.top + 14, g.left + 25 - strlen(reminder), reminder);
281:
282:     // disable color if possible
283:     if (has_colors())
284:         attroff(COLOR_PAIR(PAIR_GRID));
285: }
286:
287:
288: /*
289:  * Draws game's borders.
290:  */
291:
292: void draw_borders(void) {
293:     // get window's dimensions
294:     int maxy, maxx;
295:     getmaxyx(stdscr, maxy, maxx);
296:
297:     // enable color if possible (else b&w highlighting)
298:     if (has_colors()) {
299:         attron(A_PROTECT);
300:         attron(COLOR_PAIR(PAIR_BORDER));
301:     }
302:     else
303:         attron(A_REVERSE);
304:
305:     // draw borders
306:     for (int i = 0; i < maxx; i++) {
307:         mvaddch(0, i, ' ');
308:         mvaddch(maxy-1, i, ' ');
309:     }
310:
311:     // draw header
312:     char header[maxx+1];
313:     sprintf(header, "%s by %s", TITLE, AUTHOR);
314:     mvaddstr(0, (maxx - strlen(header)) / 2, header);
315:
316:     // draw footer
317:     mvaddstr(maxy-1, 1, "[N]ew Game    [R]estart Game    [U]ndo");
318:     mvaddstr(maxy-1, maxx-13, "[Q]uit Game");
319:
320:     // disable color if possible (else b&w highlighting)
```

```

321:     if (has_colors())
322:         attroff(COLOR_PAIR(PAIR_BORDER));
323:     else
324:         attroff(A_REVERSE);
325: }
326:
327:
328: /*
329:  * Draws game's logo. Must be called after draw_grid has been
330:  * called at least once.
331:  */
332:
333: void draw_logo(void) {
334:     // determine top-left coordinates of logo
335:     int top = g.top + 2;
336:     int left = g.left + 30;
337:
338:     // enable color if possible
339:     if (has_colors())
340:         attron(COLOR_PAIR(PAIR_LOGO));
341:
342:     // draw logo
343:     mvaddstr(top + 0, left, "                                ");
344:     mvaddstr(top + 1, left, "                                ");
345:     mvaddstr(top + 2, left, "                                ");
346:     mvaddstr(top + 3, left, "/_||_||_|/_|/_\\||_|/_||_||");
347:     mvaddstr(top + 4, left, "\\_\\_\\_||_|(|_|(|_||<|_|_|");
348:     mvaddstr(top + 5, left, "|_/_\\_\\_/_|\\_\\_/_|_|\\_\\_\\_\\_\\_\\_");
349:
350:     // sign logo
351:     char signature[3+strlen(AUTHOR)+1];
352:     sprintf(signature, "by %s", AUTHOR);
353:     mvaddstr(top + 7, left + 35 - strlen(signature) - 1, signature);
354:
355:     // disable color if possible
356:     if (has_colors())
357:         attroff(COLOR_PAIR(PAIR_LOGO));
358: }
359:
360:
361: /*
362:  * Draw's game's numbers. Must be called after draw_grid has been
363:  * called at least once.
364:  */
365:
366: void draw_numbers(void) {
367:     // iterate over board's numbers
368:     for (int i = 0; i < 9; i++) {
369:         for (int j = 0; j < 9; j++) {
370:             // determine char
371:             char c = (g.board[i][j] == 0) ? '.' : g.board[i][j] + '0';
372:
373:             // if the number came with the board, display in a different
374:             // color
375:             if (!g.won)
376:                 if (!g.changeable[i][j])
377:                     if (has_colors())
378:                         attron(COLOR_PAIR(PAIR_FIXED));
379:
380:             // if the game has been won, color all numbers green to celebrate
381:             if (g.won)
382:                 if (has_colors())
383:                     attron(COLOR_PAIR(PAIR_WON));
384:             mvaddch(g.top + i + 1 + i/3, g.left + 2 + 2*(j + j/3), c);

```

```
385:
386:     // turn the colors back off
387:     if(has_colors()) {
388:         if(g.won)
389:             attroff(COLOR_PAIR(PAIR_WON));
390:         else
391:             attroff(COLOR_PAIR(PAIR_FIXED));
392:     }
393:     refresh();
394: }
395: }
396: }
397:
398:
399: /*
400:  * Designed to handles signals (e.g., SIGWINCH).
401:  */
402:
403: void handle_signal(int signum) {
404:     // handle a change in the window (i.e., a resizing)
405:     if (signum == SIGWINCH)
406:         redraw_all();
407:
408:     // re-register myself so this signal gets handled in future too
409:     signal(signum, (void (*)(int)) handle_signal);
410: }
411:
412:
413: /*
414:  * Hides banner.
415:  */
416:
417: void hide_banner(void) {
418:     // get window's dimensions
419:     int maxy, maxx;
420:     getmaxyx(stdscr, maxy, maxx);
421:
422:     // overwrite banner with spaces
423:     for (int i = 0; i < maxx; i++)
424:         mvaddch(g.top + 16, i, ' ');
425: }
426:
427:
428: /*
429:  * Loads current board from disk, returning true iff successful.
430:  */
431:
432: bool load_board(void) {
433:     // open file with boards of specified level
434:     char filename[strlen(g.level) + 5];
435:     sprintf(filename, "%s.bin", g.level);
436:     FILE *fp = fopen(filename, "rb");
437:     if (fp == NULL)
438:         return false;
439:
440:     // determine file's size
441:     fseek(fp, 0, SEEK_END);
442:     int size = ftell(fp);
443:
444:     // ensure file is of expected size
445:     if (size % (81 * INTSIZE) != 0) {
446:         fclose(fp);
447:         return false;
448:     }
```



```
449:
450: // compute offset of specified board
451: int offset = ((g.number - 1) * 81 * INTSIZE);
452:
453: // seek to specified board
454: fseek(fp, offset, SEEK_SET);
455:
456: // read board into memory
457: if (fread(g.board, 81 * INTSIZE, 1, fp) != 1) {
458:     fclose(fp);
459:     return false;
460: }
461:
462: // copy the initial state of the board into memory
463: for(int i = 0; i < 9; i++)
464:     for(int j = 0; j < 9; j++)
465:         g.changeable[i][j] = !g.board[i][j];
466:
467: // a new board is not yet won
468: g.won = false;
469: hide_banner();
470:
471: // nothing to undo on a new board either
472: g.lastmove.y = -1;
473: g.lastmove.x = -1;
474: g.lastmove.ch = -1;
475:
476: // w00t
477: fclose(fp);
478: return true;
479: }
480:
481:
482: /*
483:  * Logs input and board's state to log.txt to facilitate automated tests.
484:  */
485:
486: void log_move(int ch) {
487:     // open log
488:     FILE *fp = fopen("log.txt", "a");
489:     if (fp == NULL)
490:         return;
491:
492:     // log input
493:     fprintf(fp, "%d\n", ch);
494:
495:     // log board
496:     for (int i = 0; i < 9; i++) {
497:         for (int j = 0; j < 9; j++)
498:             fprintf(fp, "%d", g.board[i][j]);
499:         fprintf(fp, "\n");
500:     }
501:
502:     // that's it
503:     fclose(fp);
504: }
505:
506:
507: /*
508:  * (Re)draws everything on the screen.
509:  */
510:
511: void redraw_all(void) {
512:     // reset ncurses
```

```
513:     endwin();
514:     refresh();
515:
516:     // clear screen
517:     clear();
518:
519:     // re-draw everything
520:     draw_borders();
521:     draw_grid();
522:     draw_logo();
523:     draw_numbers();
524:
525:     // show cursor
526:     show_cursor();
527: }
528:
529:
530: /*
531:  * (Re)starts current game, returning true iff succesful.
532:  */
533:
534: bool restart_game(void) {
535:     // reload current game
536:     if (!load_board())
537:         return false;
538:
539:     // redraw board
540:     draw_grid();
541:     draw_numbers();
542:
543:     // get window's dimensions
544:     int maxy, maxx;
545:     getmaxyx(stdscr, maxy, maxx);
546:
547:     // move cursor to board's center
548:     g.y = g.x = 4;
549:     show_cursor();
550:
551:     // remove log, if any
552:     remove("log.txt");
553:
554:     // w00t
555:     return true;
556: }
557:
558:
559: /*
560:  * Shows cursor at (g.y, g.x).
561:  */
562:
563: void show_cursor(void) {
564:     // restore cursor's location
565:     move(g.top + g.y + 1 + g.y/3, g.left + 2 + 2*(g.x + g.x/3));
566: }
567:
568:
569: /*
570:  * Shows a banner. Must be called after show_grid has been
571:  * called at least once.
572:  */
573:
574: void show_banner(char *b) {
575:     // enable color if possible
576:     if (has_colors())
```

```
577:     attron(COLOR_PAIR(PAIR_BANNER));
578:
579:     // determine where top-left corner of board belongs
580:     mvaddstr(g.top + 16, g.left + 64 - strlen(b), b);
581:
582:     // disable color if possible
583:     if (has_colors())
584:         attroff(COLOR_PAIR(PAIR_BANNER));
585: }
586:
587:
588: /*
589:  * Shuts down ncurses.
590:  */
591:
592: void shutdown(void) {
593:     endwin();
594: }
595:
596:
597: /*
598:  * Starts up ncurses. Returns true iff successful.
599:  */
600:
601: bool startup(void) {
602:     // initialize ncurses
603:     if (initscr() == NULL)
604:         return false;
605:
606:     // prepare for color if possible
607:     if (has_colors()) {
608:         // enable color
609:         if (start_color() == ERR || attron(A_PROTECT) == ERR) {
610:             endwin();
611:             return false;
612:         }
613:
614:         // initialize pairs of colors
615:         if (init_pair(PAIR_BANNER, FG_BANNER, BG_BANNER) == ERR ||
616:             init_pair(PAIR_GRID, FG_GRID, BG_GRID) == ERR ||
617:             init_pair(PAIR_BORDER, FG_BORDER, BG_BORDER) == ERR ||
618:             init_pair(PAIR_LOGO, FG_LOGO, BG_LOGO) == ERR ||
619:             init_pair(PAIR_FIXED, FG_FIXED, BG_FIXED) == ERR ||
620:             init_pair(PAIR_WON, FG_WON, BG_WON) == ERR) {
621:             endwin();
622:             return false;
623:         }
624:     }
625:
626:     // don't echo keyboard input
627:     if (noecho() == ERR) {
628:         endwin();
629:         return false;
630:     }
631:
632:     // disable line buffering and certain signals
633:     if (raw() == ERR) {
634:         endwin();
635:         return false;
636:     }
637:
638:     // enable arrow keys
639:     if (keypad(stdscr, true) == ERR) {
640:         endwin();
```

```
641:     return false;
642: }
643:
644: // wait 1000 ms at a time for input
645: timeout(1000);
646:
647: // w00t
648: return true;
649: }
650:
651: /*
652:  * Moves the cursor
653:  */
654: void move_cursor(int ch) {
655:     switch(ch) {
656:
657:         // decrement g.y and account for wraparound
658:         case KEY_UP:
659:             g.y = (g.y == 0) ? 8 : g.y - 1;
660:             break;
661:
662:         // increment g.y and account for wraparound
663:         case KEY_DOWN:
664:             g.y = (g.y == 8) ? 0 : g.y + 1;
665:             break;
666:
667:         // decrement g.x and account for wraparound
668:         case KEY_LEFT:
669:             g.x = (g.x == 0) ? 8 : g.x - 1;
670:             break;
671:
672:         // increment g.x and account for wraparound
673:         case KEY_RIGHT:
674:             g.x = (g.x == 8) ? 0 : g.x + 1;
675:             break;
676:     }
677:     return;
678: }
679:
680: /*
681:  * allow a user to change the value of a space on the board
682:  */
683: void insert_symbol(int ch) {
684:
685:     // we only allow moves if the initial square was blank and the board isn't
686:     // locked down because of a win
687:     if(g.changeable[g.y][g.x] && !g.won) {
688:
689:         // we also prevent the user from making a move at all, if it would be
690:         // an illegal move
691:         if(legal_move(ch - '0')) {
692:             hide_banner();
693:
694:             // save what's currently at that location, in case we need to undo
695:             g.lastmove.ch = g.board[g.y][g.x];
696:             if(isdigit(ch))
697:                 g.board[g.y][g.x] = ch - '0';
698:             else
699:                 g.board[g.y][g.x] = 0;
700:         }
701:     }
702:
703:     // Save the current cursor position in case we need to later undo this move
704:     g.lastmove.y = g.y;
```

```
705:    g.lastmove.x = g.x;
706:
707:    return;
708: }
709:
710: /*
711:  * check if a move is legal
712:  */
713: bool legal_move(int x) {
714:     return (legal_row(x) && legal_col(x) && legal_box(x));
715: }
716:
717: /*
718:  * check if a move is legal in the current box
719:  */
720: bool legal_box(int x) {
721:
722:     // calculate the upper, left-hand coordinates of the current box
723:     int top = g.y - (g.y % 3);
724:     int left = g.x - (g.x % 3);
725:
726:     // iterate through the box, skipping intended placement of the item, to
727:     // see if item already exists in the box
728:     for(int i = top; i < top + 3; i++)
729:         for(int j = left; j < left + 3; j++)
730:             if((g.board[i][j] == x) && (i != g.y) && (j != g.x)) {
731:                 show_banner("That number already appears in this box!");
732:                 return false;
733:             }
734:     return true;
735: }
736:
737: /*
738:  * check if a move is legal in the current column
739:  */
740: bool legal_col(int x) {
741:
742:     // iterate through column, skipping over intended placement of the item
743:     // to see if item already exists in column
744:     for(int i = 0; i < 9; i++)
745:         if((g.board[i][g.x] == x) && (i != g.y)) {
746:             show_banner("That number already appears in this column!");
747:             return false;
748:         }
749:     return true;
750: }
751:
752: /*
753:  * check if a move is legal in the current row
754:  */
755: bool legal_row(int x) {
756:
757:     // iterate through row, skipping over intended placement of the item to
758:     // see if item already exists in row
759:     for(int i = 0; i < 9; i++)
760:         if((g.board[g.y][i] == x) && (i != g.x)) {
761:             show_banner("That number already appears in this row!");
762:             return false;
763:         }
764:     return true;
765: }
766:
767: /*
768:  * Check for the win - only need to make sure there aren't any blank spaces
```

```
769:  * since legal_move() prevents any illegal move from being made
770:  */
771: bool game_won(void) {
772:
773:     // search for blank spaces
774:     for(int i = 0; i < 9; i++)
775:         for(int j = 0; j < 9; j++)
776:             if(g.board[i][j] == 0)
777:                 return false;
778:
779:     // lock down the board and show the banner
780:     g.won = true;
781:     show_banner("YOU WIN!");
782:     return true;
783: }
784:
785: void undo(void) {
786:
787:     // we only keep memory to move back one move
788:     if(g.lastmove.y == -1)
789:         return;
790:
791:     // restore the cursor and state of the board to just before last move
792:     g.y = g.lastmove.y;
793:     g.x = g.lastmove.x;
794:     g.board[g.y][g.x] = g.lastmove.ch;
795:
796:     // sets the lastmove struct to contain empty data, since we can only undo
797:     // one thing
798:     g.lastmove.y = -1;
799:     g.lastmove.x = -1;
800:     g.lastmove.ch = -1;
801:
802:     return;
803: }
```