```
/****************************************************************************
 * sudoku.h
 *
 * Computer Science 50
 * Problem Set 4
 *
 * Compile-time options for the game of Sudoku.
 ***************************************************************************/

// game’s author
#define AUTHOR "Doug Lloyd"

// game’s title
#define TITLE "Sudoku"

// banner’s colors
#define FG_BANNER COLOR_CYAN
#define BG_BANNER COLOR_BLACK

// grid’s colors
#define FG_GRID COLOR_WHITE
#define BG_GRID COLOR_BLACK

// border’s colors
#define FG_BORDER COLOR_WHITE
#define BG_BORDER COLOR_RED

// logo’s colors
#define FG_LOGO COLOR_WHITE
#define BG_LOGO COLOR_BLACK

// built-in numbers’ colors
#define FG_FIXED COLOR_RED
#define BG_FIXED COLOR_BLACK

// game won colors
#define FG_WON COLOR_GREEN
#define BG_WON COLOR_BLACK

// nicknames for pairs of colors
enum { PAIR_BANNER = 1, PAIR_GRID, PAIR_BORDER, PAIR_LOGO,
       PAIR_FIXED, PAIR_WON };
```

```c
/****************************************************************************
 * sudoku.c
 *
 * Computer Science 50
 * Problem Set 4
 *
 * Doug Lloyd
 * October 11, 2011
 *
 * Implements the game of Sudoku.
 * EXTRA FEATURES:
 *    - cursor wrap-around
 *    - all numbers turn green on win
 *    - numbers that came with board are red
 *    - user can undo their most recent move
 ***************************************************************************/

#include "sudoku.h"

#include <ctype.h>
#include <ncurses.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>


// macro for processing control characters
#define CTRL(x) ((x) & ~0140)

// size of each int (in bytes) in *.bin files
#define INTSIZE 4


// wrapper for our game's globals
struct {
    // the current level
    char *level;

    // the game's board
    int board[9][9];

    // holds whether initial spaces of the board are changeable
    bool changeable[9][9];

    // the board's number
    int number;

    // the board's top-left coordinates
    int top, left;

    // the cursor's current location between (0,0) and (8,8)
    int y, x;

    // the state of the game as won or not won
    bool won;

    // a structure that holds information related to the last move
    struct {
      // the position of the last change
      int y, x;

```

```
        // the value inputted at the last move
        int ch;
      } lastmove;
} g;


// provided prototypes
void draw_grid(void);
void draw_borders(void);
void draw_logo(void);
void draw_numbers(void);
void hide_banner(void);
bool load_board(void);
void handle_signal(int signum);
void log_move(int ch);
void redraw_all(void);
bool restart_game(void);
void show_banner(char *b);
void show_cursor(void);
void shutdown(void);
bool startup(void);

// added prototypes
void move_cursor(int ch);
void insert_symbol(int ch);
bool legal_move(int x);
bool legal_box(int x);
bool legal_col(int x);
bool legal_row(int x);
bool game_won(void);
void undo(void);

/*
 * Main driver for the game.
 */

int main(int argc, char *argv[]) {
  // define usage
  const char *usage = "Usage: sudoku n00b|l33t [#]\n";

  // ensure that number of arguments is as expected
  if (argc != 2 && argc != 3) {
    fprintf(stderr, usage);
    return 1;
  }

  // ensure that level is valid
  if (strcmp(argv[1], "debug") == 0)
    g.level = "debug";
  else if (strcmp(argv[1], "n00b") == 0)
    g.level = "n00b";
  else if (strcmp(argv[1], "l33t") == 0)
    g.level = "l33t";
  else {
    fprintf(stderr, usage);
    return 2;
  }

  // n00b and l33t levels have 1024 boards; debug level has 9
  int max = (strcmp(g.level, "debug") == 0) ? 9 : 1024;

  // ensure that #, if provided, is in [1, max]
  if (argc == 3) {
    // ensure n is integral
```

```c
    char c;
    if (sscanf(argv[2], " %d %c", &g.number, &c) != 1) {
      fprintf(stderr, usage);
      return 3;
    }

    // ensure n is in [1, max]
    if (g.number < 1 || g.number > max) {
      fprintf(stderr, "That board # does not exist!\n");
      return 4;
    }

    // seed PRNG with # so that we get same sequence of boards
    srand(g.number);
  }
  else {
    // seed PRNG with current time so that we get any sequence of boards
    srand(time(NULL));

    // choose a random n in [1, max]
    g.number = rand() % max + 1;
  }

  // start up ncurses
  if (!startup()) {
    fprintf(stderr, "Error starting up ncurses!\n");
    return 5;
  }

  // register handler for SIGWINCH (SIGnal WINdow CHanged)
  signal(SIGWINCH, (void (*)(int)) handle_signal);

  // start the first game
  if (!restart_game()) {
    shutdown();
    fprintf(stderr, "Could not load board from disk!\n");
    return 6;
  }
  redraw_all();

  // let the user play!
  int ch;
  do {
    // refresh the screen
    refresh();

    // get user’s input
    ch = getch();

    // capitalize input to simplify cases
    ch = toupper(ch);

    // process user’s input
    switch (ch) {

      // start a new game
      case ’N’:
        g.number = rand() % max + 1;
        if (!restart_game()) {
          shutdown();
          fprintf(stderr, "Could not load board from disk!\n");
          return 6;
        }
        break;

      // restart current game
      case 'R':
        if (!restart_game()) {
          shutdown();
          fprintf(stderr, "Could not load board from disk!\n");
          return 6;
        }
        break;

      // let user manually redraw screen with ctrl-L
      case CTRL('l'):
        redraw_all();
        break;

      // allow for cursor movement
      case KEY_UP: case KEY_DOWN:
      case KEY_LEFT: case KEY_RIGHT:
        move_cursor(ch);
        show_cursor();
        break;

      // allow for changing of board values
      case '1': case '2': case '3': case '4': case '5':
      case '6': case '7': case '8': case '9': case '0':
      case '.': case KEY_BACKSPACE: case KEY_DC:
        insert_symbol(ch);
        game_won();
        draw_numbers();
        show_cursor();
        break;

      // let the user undo the last thing they did
      case 'U': case CTRL('z'):
        undo();
        draw_numbers();
        show_cursor();
        break;
    }

    // log input (and board's state) if any was received this iteration
    if (ch != ERR)
      log_move(ch);
  } while (ch != 'Q');

  // shut down ncurses
  shutdown();

  // tidy up the screen (using ANSI escape sequences)
  printf("\033[2J");
  printf("\033[%d;%dH", 0, 0);

  // that's all folks
  printf("\nkthxbai!\n\n");
  return 0;
}


/*
 * Draw's the game's board.
 */

void draw_grid(void) {
  // get window's dimensions
  int maxy, maxx;
  getmaxyx(stdscr, maxy, maxx);

  // determine where top-left corner of board belongs
  g.top = maxy/2 - 7;
  g.left = maxx/2 - 30;

  // enable color if possible
  if (has_colors())
    attron(COLOR_PAIR(PAIR_GRID));

  // print grid
  for (int i = 0 ; i < 3 ; ++i ) {
    mvaddstr(g.top + 0 + 4 * i, g.left, "+-------+-------+-------+");
    mvaddstr(g.top + 1 + 4 * i, g.left, "|       |       |       |");
    mvaddstr(g.top + 2 + 4 * i, g.left, "|       |       |       |");
    mvaddstr(g.top + 3 + 4 * i, g.left, "|       |       |       |");
  }
  mvaddstr(g.top + 4 * 3, g.left, "+-------+-------+-------+" );

  // remind user of level and #
  char reminder[maxx+1];
  sprintf(reminder, "   playing %s #%d", g.level, g.number);
  mvaddstr(g.top + 14, g.left + 25 - strlen(reminder), reminder);

  // disable color if possible
  if (has_colors())
    attroff(COLOR_PAIR(PAIR_GRID));
}


/*
 * Draws game’s borders.
 */

void draw_borders(void) {
  // get window’s dimensions
  int maxy, maxx;
  getmaxyx(stdscr, maxy, maxx);

  // enable color if possible (else b&w highlighting)
  if (has_colors()) {
    attron(A_PROTECT);
    attron(COLOR_PAIR(PAIR_BORDER));
  }
  else
    attron(A_REVERSE);

    // draw borders
  for (int i = 0; i < maxx; i++) {
    mvaddch(0, i, ’ ’);
    mvaddch(maxy-1, i, ’ ’);
  }

  // draw header
  char header[maxx+1];
  sprintf(header, "%s by %s", TITLE, AUTHOR);
  mvaddstr(0, (maxx - strlen(header)) / 2, header);

  // draw footer
  mvaddstr(maxy-1, 1, "[N]ew Game   [R]estart Game   [U]ndo");
  mvaddstr(maxy-1, maxx-13, "[Q]uit Game");

  // disable color if possible (else b&w highlighting)
  if (has_colors())
    attroff(COLOR_PAIR(PAIR_BORDER));
  else
    attroff(A_REVERSE);
}


/*
 * Draws game’s logo.  Must be called after draw_grid has been
 * called at least once.
 */

void draw_logo(void) {
  // determine top-left coordinates of logo
  int top = g.top + 2;
  int left = g.left + 30;

  // enable color if possible
  if (has_colors())
    attron(COLOR_PAIR(PAIR_LOGO));

  // draw logo
  mvaddstr(top + 0, left, "               _       _          ");
  mvaddstr(top + 1, left, "              | |     | |         ");
  mvaddstr(top + 2, left, " ___ _   _  __| | ___ | | ___   _ ");
  mvaddstr(top + 3, left, "/ __| | | |/ _‘ |/ _ \\| |/ / | | |");
  mvaddstr(top + 4, left, "\\__ \\ |_| | (_| | (_) |   <| |_| |");
  mvaddstr(top + 5, left, "|___/\\__,_|\\__,_|\\___/|_|\\_\\\\__,_|");

  // sign logo
  char signature[3+strlen(AUTHOR)+1];
  sprintf(signature, "by %s", AUTHOR);
  mvaddstr(top + 7, left + 35 - strlen(signature) - 1, signature);

  // disable color if possible
  if (has_colors())
    attroff(COLOR_PAIR(PAIR_LOGO));
}


/*
 * Draw’s game’s numbers.  Must be called after draw_grid has been
 * called at least once.
 */

void draw_numbers(void) {
  // iterate over board’s numbers
  for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 9; j++) {
      // determine char
      char c = (g.board[i][j] == 0) ? ’.’ : g.board[i][j] + ’0’;

      // if the number came with the board, display in a different
      // color
      if(!g.won)
        if(!g.changeable[i][j])
          if(has_colors())
            attron(COLOR_PAIR(PAIR_FIXED));

      // if the game has been won, color all numbers green to celebrate
      if(g.won)
        if(has_colors())
          attron(COLOR_PAIR(PAIR_WON));
      mvaddch(g.top + i + 1 + i/3, g.left + 2 + 2*(j + j/3), c);
```

```c

      // turn the colors back off
      if(has_colors()) {
        if(g.won)
          attroff(COLOR_PAIR(PAIR_WON));
        else
          attroff(COLOR_PAIR(PAIR_FIXED));
      }
      refresh();
    }
  }
}


/*
 * Designed to handles signals (e.g., SIGWINCH).
 */

void handle_signal(int signum) {
  // handle a change in the window (i.e., a resizing)
  if (signum == SIGWINCH)
    redraw_all();

  // re-register myself so this signal gets handled in future too
  signal(signum, (void (*)(int)) handle_signal);
}


/*
 * Hides banner.
 */

void hide_banner(void) {
  // get window’s dimensions
  int maxy, maxx;
  getmaxyx(stdscr, maxy, maxx);

  // overwrite banner with spaces
  for (int i = 0; i < maxx; i++)
    mvaddch(g.top + 16, i, ’ ’);
}


/*
 * Loads current board from disk, returning true iff successful.
 */

bool load_board(void) {
  // open file with boards of specified level
  char filename[strlen(g.level) + 5];
  sprintf(filename, "%s.bin", g.level);
  FILE *fp = fopen(filename, "rb");
  if (fp == NULL)
    return false;

  // determine file’s size
  fseek(fp, 0, SEEK_END);
  int size = ftell(fp);

  // ensure file is of expected size
  if (size % (81 * INTSIZE) != 0) {
    fclose(fp);
    return false;
  }

  // compute offset of specified board
  int offset = ((g.number - 1) * 81 * INTSIZE);

  // seek to specified board
  fseek(fp, offset, SEEK_SET);

  // read board into memory
  if (fread(g.board, 81 * INTSIZE, 1, fp) != 1) {
    fclose(fp);
    return false;
  }

  // copy the initial state of the board into memory
  for(int i = 0; i < 9; i++)
    for(int j = 0; j < 9; j++)
      g.changeable[i][j] = !g.board[i][j];

  // a new board is not yet won
  g.won = false;
  hide_banner();

  // nothing to undo on a new board either
  g.lastmove.y = -1;
  g.lastmove.x = -1;
  g.lastmove.ch = -1;

  // w00t
  fclose(fp);
  return true;
}


/*
 * Logs input and board’s state to log.txt to facilitate automated tests.
 */

void log_move(int ch) {
  // open log
  FILE *fp = fopen("log.txt", "a");
  if (fp == NULL)
    return;

    // log input
  fprintf(fp, "%d\n", ch);

  // log board
  for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 9; j++)
      fprintf(fp, "%d", g.board[i][j]);
    fprintf(fp, "\n");
  }

  // that’s it
  fclose(fp);
}


/*
 * (Re)draws everything on the screen.
 */

void redraw_all(void) {
  // reset ncurses
```

```c
  endwin();
  refresh();

  // clear screen
  clear();

  // re-draw everything
  draw_borders();
  draw_grid();
  draw_logo();
  draw_numbers();

  // show cursor
  show_cursor();
}


/*
 * (Re)starts current game, returning true iff succesful.
 */

bool restart_game(void) {
  // reload current game
  if (!load_board())
    return false;

  // redraw board
  draw_grid();
  draw_numbers();

  // get window’s dimensions
  int maxy, maxx;
  getmaxyx(stdscr, maxy, maxx);

  // move cursor to board’s center
  g.y = g.x = 4;
  show_cursor();

  // remove log, if any
  remove("log.txt");

  // w00t
  return true;
}


/*
 * Shows cursor at (g.y, g.x).
 */

void show_cursor(void) {
  // restore cursor’s location
  move(g.top + g.y + 1 + g.y/3, g.left + 2 + 2*(g.x + g.x/3));
}


/*
 * Shows a banner.  Must be called after show_grid has been
 * called at least once.
 */

void show_banner(char *b) {
  // enable color if possible
  if (has_colors())
```

```
    attron(COLOR_PAIR(PAIR_BANNER));

  // determine where top-left corner of board belongs
  mvaddstr(g.top + 16, g.left + 64 - strlen(b), b);

  // disable color if possible
  if (has_colors())
    attroff(COLOR_PAIR(PAIR_BANNER));
}


/*
 * Shuts down ncurses.
 */

void shutdown(void) {
  endwin();
}


/*
 * Starts up ncurses.  Returns true iff successful.
 */

bool startup(void) {
  // initialize ncurses
  if (initscr() == NULL)
    return false;

  // prepare for color if possible
  if (has_colors()) {
    // enable color
    if (start_color() == ERR || attron(A_PROTECT) == ERR) {
      endwin();
      return false;
    }

    // initialize pairs of colors
    if (init_pair(PAIR_BANNER, FG_BANNER, BG_BANNER) == ERR ||
        init_pair(PAIR_GRID, FG_GRID, BG_GRID) == ERR ||
        init_pair(PAIR_BORDER, FG_BORDER, BG_BORDER) == ERR ||
        init_pair(PAIR_LOGO, FG_LOGO, BG_LOGO) == ERR ||
        init_pair(PAIR_FIXED, FG_FIXED, BG_FIXED) == ERR ||
        init_pair(PAIR_WON, FG_WON, BG_WON) == ERR) {
      endwin();
      return false;
    }
  }

  // don’t echo keyboard input
  if (noecho() == ERR) {
    endwin();
    return false;
  }

  // disable line buffering and certain signals
  if (raw() == ERR) {
    endwin();
    return false;
  }

  // enable arrow keys
  if (keypad(stdscr, true) == ERR) {
    endwin();
    return false;
  }

  // wait 1000 ms at a time for input
  timeout(1000);

  // w00t
  return true;
}

/*
 * Moves the cursor
 */
void move_cursor(int ch) {
  switch(ch) {

    // decrement g.y and account for wraparound
    case KEY_UP:
      g.y = (g.y == 0) ? 8 : g.y - 1;
      break;

    // increment g.y and account for wraparound
    case KEY_DOWN:
      g.y = (g.y == 8) ? 0 : g.y + 1;
      break;

    // decrement g.x and account for wraparound
    case KEY_LEFT:
      g.x = (g.x == 0) ? 8 : g.x - 1;
      break;

    // increment g.x and account for wraparound
    case KEY_RIGHT:
      g.x = (g.x == 8) ? 0 : g.x + 1;
      break;
  }
  return;
}

/*
 * allow a user to change the value of a space on the board
 */
void insert_symbol(int ch) {

  // we only allow moves if the initial square was blank and the board isn’t
  // locked down because of a win
  if(g.changeable[g.y][g.x] && !g.won) {

    // we also prevent the user from making a move at all, if it would be
    // an illegal move
    if(legal_move(ch - ’0’)) {
      hide_banner();

      // save what’s currently at that location, in case we need to undo
      g.lastmove.ch = g.board[g.y][g.x];
      if(isdigit(ch))
        g.board[g.y][g.x] = ch - ’0’;
      else
        g.board[g.y][g.x] = 0;
    }
  }

  // Save the current cursor position in case we need to later undo this move
  g.lastmove.y = g.y;
  g.lastmove.x = g.x;

  return;
}

/*
 * check if a move is legal
 */
bool legal_move(int x) {
  return (legal_row(x) && legal_col(x) && legal_box(x));
}

/*
 * check if a move is legal in the current box
 */
bool legal_box(int x) {

  // calcluate the upper, left-hand coordinates of the current box
  int top = g.y - (g.y % 3);
  int left = g.x - (g.x % 3);

  // iterate through the box, skipping intended placement of the item, to
  // see if item already exists in the box
  for(int i = top; i < top + 3; i++)
    for(int j = left; j < left + 3; j++)
      if((g.board[i][j] == x) && (i != g.y) && (j != g.x)) {
        show_banner("That number already appears in this box!");
        return false;
      }
  return true;
}

/*
 * check if a move is legal in the current column
 */
bool legal_col(int x) {

  // iterate through column, skipping over intended placement of the item
  // to see if item already exists in column
  for(int i = 0; i < 9; i++)
    if((g.board[i][g.x] == x) && (i != g.y)) {
      show_banner("That number already appears in this column!");
      return false;
    }
  return true;
}

/*
 * check if a move is legal in the current row
 */
bool legal_row(int x) {

  // iterate through row, skipping over intended placement of the item to
  // see if item already exists in row
  for(int i = 0; i < 9; i++)
    if((g.board[g.y][i] == x) && (i != g.x)) {
      show_banner("That number already appears in this row!");
      return false;
    }
  return true;
}

/*
 * Check for the win - only need to make sure there aren’t any blank spaces
 * since legal_move() prevents any illegal move from being made
 */
bool game_won(void) {

  // search for blank spaces
  for(int i = 0; i < 9; i++)
    for(int j = 0; j < 9; j++)
      if(g.board[i][j] == 0)
        return false;

  // lock down the board and show the banner
  g.won = true;
  show_banner("YOU WIN!");
  return true;
}

void undo(void) {

  // we only keep memory to move back one move
  if(g.lastmove.y == -1)
    return;

  // restore the cursor and state of the board to just before last move
  g.y = g.lastmove.y;
  g.x = g.lastmove.x;
  g.board[g.y][g.x] = g.lastmove.ch;

  // sets the lastmove struct to contain empty data, since we can only undo
  // one thing
  g.lastmove.y = -1;
  g.lastmove.x = -1;
  g.lastmove.ch = -1;

  return;
}
```