← prev contents next →

A text editor

Insert ordinary characters

Let's begin by writing a function that inserts a single character into an `erow`, at a given position.

kilo.c **Step 101** `row-insert-char`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row operations ***/

int editorRowCxToRx(erow *row, int cx) { … }

void editorUpdateRow(erow *row) { … }

void editorAppendRow(char *s, size_t len) { … }

void editorRowInsertChar(erow *row, int at, int c) {
  if (at < 0 || at > row->size) at = row->size;
  row->chars = realloc(row->chars, row->size + 2);
  memmove(&row->chars[at + 1], &row->chars[at], row->size - at + 1);
  row->size++;
  row->chars[at] = c;
  editorUpdateRow(row);
}

/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

compiles, but with no observable effects

`memmove()` comes from `<string.h>`. It is like `memcpy()`, but is safe to use when the source and destination arrays overlap.

First we validate `at`, which is the index we want to insert the character into. Notice that `at` is allowed to go one character past the end of the string, in which case the character should be inserted at the end of the string.

Then we allocate one more byte for the `chars` of the `erow` (we add `2` because we also have to make room for the null byte), and use `memmove()` to make room for the new

character. We increment the `size` of the `chars` array, and then actually assign the character to its position in the array. Finally, we call `editorUpdateRow()` so that the `render` and `rsize` fields get updated with the new row content.

Now we'll create a new section called `/*** editor operations ***/`. This section will contain functions that we'll call from `editorProcessKeypress()` when we're mapping keypresses to various text editing operations. We'll add a function to this section called `editorInsertChar()` which will take a character and use `editorRowInsertChar()` to insert that character into the position that the cursor is at.

kilo.c **Step 102** `editor-insert-char`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row operations ***/

int editorRowCxToRx(erow *row, int cx) { … }

void editorUpdateRow(erow *row) { … }

void editorAppendRow(char *s, size_t len) { … }

void editorRowInsertChar(erow *row, int at, int c) { … }

/*** editor operations ***/

void editorInsertChar(int c) {
  if (E.cy == E.numrows) {
    editorAppendRow("", 0);
  }
  editorRowInsertChar(&E.row[E.cy], E.cx, c);
  E.cx++;
}

/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

compiles, but with no observable effects

If `E.cy == E.numrows`, then the cursor is on the tilde line after the end of the file, so we need to append a new row to the file before inserting a character there. After inserting a character, we move the cursor forward so that the next character the user inserts will go after the character just inserted.

Notice that `editorInsertChar()` doesn't have to worry about the details of

modifying an `erow`, and `editorRowInsertChar()` doesn't have to worry about where the cursor is. That is the difference between functions in the `/*** editor operations ***/` section and functions in the `/*** row operations ***/` section.

Let's call `editorInsertChar()` in the `default:` case of the `switch` statement in `editorProcessKeypress()`. This will allow any keypress that isn't mapped to another editor function to be inserted directly into the text being edited.

kilo.c **Step 103** `key-insert-char`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/

void editorMoveCursor(int key) { … }

void editorProcessKeypress() {
  int c = editorReadKey();

  switch (c) {
    case CTRL_KEY('q'):
      write(STDOUT_FILENO, "\x1b[2J", 4);
      write(STDOUT_FILENO, "\x1b[H", 3);
      exit(0);
      break;

    case HOME_KEY:
      E.cx = 0;
      break;

    case END_KEY:
      if (E.cy < E.numrows)
        E.cx = E.row[E.cy].size;
      break;

    case PAGE_UP:
    case PAGE_DOWN:
      {
        if (c == PAGE_UP) {
          E.cy = E.rowoff;
        } else if (c == PAGE_DOWN) {
          E.cy = E.rowoff + E.screenrows - 1;
          if (E.cy > E.numrows) E.cy = E.numrows;
        }
```

```
        int times = E.screenrows;
        while (times--)
          editorMoveCursor(c == PAGE_UP ? ARROW_UP : ARROW_DOWN);
      }
      break;

    case ARROW_UP:
    case ARROW_DOWN:
    case ARROW_LEFT:
    case ARROW_RIGHT:
      editorMoveCursor(c);
      break;

    default:
      editorInsertChar(c);
      break;
  }
}

/*** init ***/
```

⤢ compiles

We've now officially upgraded our text viewer to a text editor.

Prevent inserting special characters

Currently, if you press keys like `Backspace` or `Enter`, those characters will be inserted directly into the text, which we certainly don't want. Let's handle a bunch of these special keys in `editorProcessKeypress()`, so that they don't fall through to the `default` case of calling `editorInsertChar()`.

kilo.c **Step 104** `block-special-chars`

```
/*** includes ***/
/*** defines ***/

#define KILO_VERSION "0.0.1"
#define KILO_TAB_STOP 8

#define CTRL_KEY(k) ((k) & 0x1f)

enum editorKey {
  BACKSPACE = 127,
  ARROW_LEFT = 1000,
  ARROW_RIGHT,
  ARROW_UP,
  ARROW_DOWN,
  DEL_KEY,
  HOME_KEY,
  END_KEY,
```

```
  PAGE_UP,
  PAGE_DOWN
};

/*** data ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/

void editorMoveCursor(int key) { … }

void editorProcessKeypress() {
  int c = editorReadKey();

  switch (c) {
    case '\r':
      /* TODO */
      break;

    case CTRL_KEY('q'):
      write(STDOUT_FILENO, "\x1b[2J", 4);
      write(STDOUT_FILENO, "\x1b[H", 3);
      exit(0);
      break;

    case HOME_KEY:
      E.cx = 0;
      break;

    case END_KEY:
      if (E.cy < E.numrows)
        E.cx = E.row[E.cy].size;
      break;

    case BACKSPACE:
    case CTRL_KEY('h'):
    case DEL_KEY:
      /* TODO */
      break;

    case PAGE_UP:
    case PAGE_DOWN:
      {
        if (c == PAGE_UP) {
          E.cy = E.rowoff;
        } else if (c == PAGE_DOWN) {
          E.cy = E.rowoff + E.screenrows - 1;
          if (E.cy > E.numrows) E.cy = E.numrows;
        }
```

```
        int times = E.screenrows;
        while (times--)
          editorMoveCursor(c == PAGE_UP ? ARROW_UP : ARROW_DOWN);
      }
      break;

    case ARROW_UP:
    case ARROW_DOWN:
    case ARROW_LEFT:
    case ARROW_RIGHT:
      editorMoveCursor(c);
      break;

    case CTRL_KEY('l'):
    case '\x1b':
      break;

    default:
      editorInsertChar(c);
      break;
  }
}

/*** init ***/
```

compiles

`Backspace` doesn't have a human-readable backslash-escape representation in C (like `\n`, `\r`, and so on), so we make it part of the `editorKey` enum and assign it its ASCII value of `127`.

In `editorProcessKeypress()`, the first new key we add to the `switch` statement is `'\r'`, which is the `Enter` key. For now we want to ignore it, but obviously we'll be making it do something later, so we mark it with a `TODO` comment.

We handle `Backspace` and `Delete` in a similar way, marking them with a `TODO`. We also handle the `Ctrl-H` key combination, which sends the control code `8`, which is originally what the `Backspace` character would send back in the day. If you look at the ASCII table, you'll see that ASCII code `8` is named `BS` for "backspace", and ASCII code `127` is named `DEL` for "delete". But for whatever reason, in modern computers the `Backspace` key is mapped to `127` and the `Delete` key is mapped to the escape sequence `<esc>[3~`, as we saw at the end of chapter 3.

Lastly, we handle `Ctrl-L` and `Escape` by not doing anything when those keys are pressed. `Ctrl-L` is traditionally used to refresh the screen in terminal programs. In our text editor, the screen refreshes after *any* keypress, so we don't have to do anything else to implement that feature. We ignore the `Escape` key because there are many key

escape sequences that we aren't handling (such as the `F1`–`F12` keys), and the way we wrote `editorReadKey()`, pressing those keys will be equivalent to pressing the `Escape` key. We don't want the user to unwittingly insert the escape character `27` into their text, so we ignore those keypresses.

Save to disk

Now that we've finally made text editable, let's implement saving to disk. First we'll write a function that converts our array of `erow` structs into a single string that is ready to be written out to a file.

kilo.c **Step 105** rows-to-string

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/

char *editorRowsToString(int *buflen) {
  int totlen = 0;
  int j;
  for (j = 0; j < E.numrows; j++)
    totlen += E.row[j].size + 1;
  *buflen = totlen;

  char *buf = malloc(totlen);
  char *p = buf;
  for (j = 0; j < E.numrows; j++) {
    memcpy(p, E.row[j].chars, E.row[j].size);
    p += E.row[j].size;
    *p = '\n';
    p++;
  }

  return buf;
}

void editorOpen(char *filename) { … }

/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

compiles, but with no observable effects

First we add up the lengths of each row of text, adding `1` to each one for the newline character we'll add to the end of each line. We save the total length into `buflen`, to

tell the caller how long the string is.

Then, after allocating the required memory, we loop through the rows, and `memcpy()` the contents of each row to the end of the buffer, appending a newline character after each row.

We return `buf`, expecting the caller to `free()` the memory.

Now we'll implement the `editorSave()` function, which will actually write the string returned by `editorRowsToString()` to disk.

kilo.c **Step 106** save

```
/*** includes ***/

#define _DEFAULT_SOURCE
#define _BSD_SOURCE
#define _GNU_SOURCE

#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <termios.h>
#include <time.h>
#include <unistd.h>

/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/

char *editorRowsToString(int *buflen) { … }

void editorOpen(char *filename) { … }

void editorSave() {
  if (E.filename == NULL) return;

  int len;
  char *buf = editorRowsToString(&len);

  int fd = open(E.filename, O_RDWR | O_CREAT, 0644);
  ftruncate(fd, len);
  write(fd, buf, len);
```

```
  close(fd);
  free(buf);
}

/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

compiles, but with no observable effects

`open()`, `O_RDWR`, and `O_CREAT` come from `<fcntl.h>`. `ftruncate()` and `close()` come from `<unistd.h>`.

If it's a new file, then `E.filename` will be `NULL`, and we won't know where to save the file, so we just `return` without doing anything for now. Later, we'll figure out how to prompt the user for a filename.

Otherwise, we call `editorRowsToString()`, and `write()` the string to the path in `E.filename`. We tell `open()` we want to create a new file if it doesn't already exist (`O_CREAT`), and we want to open it for reading and writing (`O_RDWR`). Because we used the `O_CREAT` flag, we have to pass an extra argument containing the mode (the permissions) the new file should have. `0644` is the standard permissions you usually want for text files. It gives the owner of the file permission to read and write the file, and everyone else only gets permission to read the file.

`ftruncate()` sets the file's size to the specified length. If the file is larger than that, it will cut off any data at the end of the file to make it that length. If the file is shorter, it will add `0` bytes at the end to make it that length.

The normal way to overwrite a file is to pass the `O_TRUNC` flag to `open()`, which truncates the file completely, making it an empty file, before writing the new data into it. By truncating the file ourselves to the same length as the data we are planning to write into it, we are making the whole overwriting operation a little bit safer in case the `ftruncate()` call succeeds but the `write()` call fails. In that case, the file would still contain most of the data it had before. But if the file was truncated completely by the `open()` call and then the `write()` failed, you'd end up with all of your data lost.

More advanced editors will write to a new, temporary file, and then rename that file to the actual file the user wants to overwrite, and they'll carefully check for errors through the whole process.

Anyways, all we have to do now is map a key to `editorSave()`, so let's do it! We'll use `Ctrl-S`.

kilo.c **Step 107** `ctrl-s`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/

void editorMoveCursor(int key) { … }

void editorProcessKeypress() {
  int c = editorReadKey();

  switch (c) {
    case '\r':
      /* TODO */
      break;

    case CTRL_KEY('q'):
      write(STDOUT_FILENO, "\x1b[2J", 4);
      write(STDOUT_FILENO, "\x1b[H", 3);
      exit(0);
      break;

    case CTRL_KEY('s'):
      editorSave();
      break;

    case HOME_KEY:
      E.cx = 0;
      break;

    case END_KEY:
      if (E.cy < E.numrows)
        E.cx = E.row[E.cy].size;
      break;

    case BACKSPACE:
    case CTRL_KEY('h'):
    case DEL_KEY:
      /* TODO */
      break;

    case PAGE_UP:
    case PAGE_DOWN:
      {
        if (c == PAGE_UP) {
          E.cy = E.rowoff;
        } else if (c == PAGE_DOWN) {
          E.cy = E.rowoff + E.screenrows - 1;
```

```
          if (E.cy > E.numrows) E.cy = E.numrows;
        }

        int times = E.screenrows;
        while (times--)
          editorMoveCursor(c == PAGE_UP ? ARROW_UP : ARROW_DOWN);
      }
      break;

    case ARROW_UP:
    case ARROW_DOWN:
    case ARROW_LEFT:
    case ARROW_RIGHT:
      editorMoveCursor(c);
      break;

    case CTRL_KEY('l'):
    case '\x1b':
      break;

    default:
      editorInsertChar(c);
      break;
  }
}

/*** init ***/
```

⤢ compiles

You should be able to open a file in the editor, insert some characters, press `Ctrl-S`, and reopen the file to confirm that the changes you made were saved.

Let's add error handling to `editorSave()`.

kilo.c **Step 108** `save-errors`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/

char *editorRowsToString(int *buflen) { … }

void editorOpen(char *filename) { … }

void editorSave() {
  if (E.filename == NULL) return;

  int len;
  char *buf = editorRowsToString(&len);
```

```
  int fd = open(E.filename, O_RDWR | O_CREAT, 0644);
  if (fd != -1) {
    if (ftruncate(fd, len) != -1) {
      if (write(fd, buf, len) == len) {
        close(fd);
        free(buf);
        return;
      }
    }
    close(fd);
  }

  free(buf);
}

/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

⤢ compiles

`open()` and `ftruncate()` both return `-1` on error. We expect `write()` to return the number of bytes we told it to write. Whether or not an error occurred, we ensure that the file is closed and the memory that `buf` points to is freed.

Let's use `editorSetStatusMessage()` to notify the user whether the save succeeded or not. While we're at it, we'll add the `Ctrl-S` key binding to the help message that's set in `main()`.

kilo.c **Step 109** `save-status-message`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/

char *editorRowsToString(int *buflen) { … }

void editorOpen(char *filename) { … }

void editorSave() {
  if (E.filename == NULL) return;

  int len;
  char *buf = editorRowsToString(&len);

  int fd = open(E.filename, O_RDWR | O_CREAT, 0644);
  if (fd != -1) {
```

```
    if (ftruncate(fd, len) != -1) {
      if (write(fd, buf, len) == len) {
        close(fd);
        free(buf);
        editorSetStatusMessage("%d bytes written to disk", len);
        return;
      }
    }
    close(fd);
  }

  free(buf);
  editorSetStatusMessage("Can't save! I/O error: %s", strerror(errno));
}

/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/

void initEditor() { … }

int main(int argc, char *argv[]) {
  enableRawMode();
  initEditor();
  if (argc >= 2) {
    editorOpen(argv[1]);
  }

  editorSetStatusMessage("HELP: Ctrl-S = save | Ctrl-Q = quit");

  while (1) {
    editorRefreshScreen();
    editorProcessKeypress();
  }

  return 0;
}
```

♏ doesn't compile

`strerror()` comes from `<string.h>`.

`strerror()` is like `perror()` (which we use in `die()`), but it takes the `errno` value as an argument and returns the human-readable string for that error code, so that we can make the error a part of the status message we display to the user.

The above code doesn't actually compile, because we are trying to call `editorSetStatusMessage()` before it is defined in the file. You can't do that in C, because C was made to be a language that can be compiled in a single pass, meaning it should be possible to compile each part of a program without knowing what comes later in the program.

When we call a function in C, the compiler needs to know the arguments and return value of that function. We can tell the compiler this information about `editorSetStatusMessage()` by declaring a function prototype for it near the top of the file. This allows us to call the function before it is defined. We'll add a new `/*** prototypes ***/` section and put the declaration under it.

kilo.c **Step 110** `prototypes`

```
/*** includes ***/
/*** defines ***/
/*** data ***/

typedef struct erow { … } erow;

struct editorConfig { … };

struct editorConfig E;

/*** prototypes ***/

void editorSetStatusMessage(const char *fmt, ...);

/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

compiles

Dirty flag

We'd like to keep track of whether the text loaded in our editor differs from what's in the file. Then we can warn the user that they might lose unsaved changes when they try to quit.

We call a text buffer "dirty" if it has been modified since opening or saving the file. Let's add a `dirty` variable to the global editor state, and initialize it to `0`.

kilo.c **Step 111** `dirty`

```
/*** includes ***/
/*** defines ***/
/*** data ***/

typedef struct erow { … } erow;

struct editorConfig {
  int cx, cy;
```

```
  int rx;
  int rowoff;
  int coloff;
  int screenrows;
  int screencols;
  int numrows;
  erow *row;
  int dirty;
  char *filename;
  char statusmsg[80];
  time_t statusmsg_time;
  struct termios orig_termios;
};

struct editorConfig E;

/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/

void initEditor() {
  E.cx = 0;
  E.cy = 0;
  E.rx = 0;
  E.rowoff = 0;
  E.coloff = 0;
  E.numrows = 0;
  E.row = NULL;
  E.dirty = 0;
  E.filename = NULL;
  E.statusmsg[0] = '\0';
  E.statusmsg_time = 0;

  if (getWindowSize(&E.screenrows, &E.screencols) == -1) die("getWindowSize");
  E.screenrows -= 2;
}

int main(int argc, char *argv[]) { … }
```

compiles, but with no observable effects

Let's show the state of `E.dirty` in the status bar, by displaying `(modified)` after the filename if the file has been modified.

kilo.c **Step 112** `show-dirty`

```
/*** includes ***/
/*** defines ***/
```

```
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/

void editorScroll() { … }

void editorDrawRows(struct abuf *ab) { … }

void editorDrawStatusBar(struct abuf *ab) {
  abAppend(ab, "\x1b[7m", 4);
  char status[80], rstatus[80];
  int len = snprintf(status, sizeof(status), "%.20s - %d lines %s",
    E.filename ? E.filename : "[No Name]", E.numrows,
    E.dirty ? "(modified)" : "");
  int rlen = snprintf(rstatus, sizeof(rstatus), "%d/%d",
    E.cy + 1, E.numrows);
  if (len > E.screencols) len = E.screencols;
  abAppend(ab, status, len);
  while (len < E.screencols) {
    if (E.screencols - len == rlen) {
      abAppend(ab, rstatus, rlen);
      break;
    } else {
      abAppend(ab, " ", 1);
      len++;
    }
  }
  abAppend(ab, "\x1b[m", 3);
  abAppend(ab, "\r\n", 2);
}

void editorDrawMessageBar(struct abuf *ab) { … }

void editorRefreshScreen() { … }

void editorSetStatusMessage(const char *fmt, ...) { … }

/*** input ***/
/*** init ***/
```

compiles, but with no observable effects

Now let's increment `E.dirty` in each row operation that makes a change to the text.

kilo.c **Step 113** `increment-dirty`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
```

```
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/

int editorRowCxToRx(erow *row, int cx) { … }

void editorUpdateRow(erow *row) { … }

void editorAppendRow(char *s, size_t len) {
  E.row = realloc(E.row, sizeof(erow) * (E.numrows + 1));

  int at = E.numrows;
  E.row[at].size = len;
  E.row[at].chars = malloc(len + 1);
  memcpy(E.row[at].chars, s, len);
  E.row[at].chars[len] = '\0';

  E.row[at].rsize = 0;
  E.row[at].render = NULL;
  editorUpdateRow(&E.row[at]);

  E.numrows++;
  E.dirty++;
}

void editorRowInsertChar(erow *row, int at, int c) {
  if (at < 0 || at > row->size) at = row->size;
  row->chars = realloc(row->chars, row->size + 2);
  memmove(&row->chars[at + 1], &row->chars[at], row->size - at + 1);
  row->size++;
  row->chars[at] = c;
  editorUpdateRow(row);
  E.dirty++;
}

/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

⤢ compiles

We could have used `E.dirty = 1` instead of `E.dirty++`, but by incrementing it we can have a sense of "how dirty" the file is, which could be useful. (We'll just be treating `E.dirty` as a boolean value in this tutorial, so it doesn't really matter.)

If you open a file at this point, you'll see that `(modified)` appears right away, before you make any changes. That's because `editorOpen()` calls `editorAppendRow()`, which increments `E.dirty`. To fix that, let's reset `E.dirty` to `0` at the end of `editorOpen()`, and also in `editorSave()`.

kilo.c **Step 114** reset-dirty

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/

char *editorRowsToString(int *buflen) { … }

void editorOpen(char *filename) {
  free(E.filename);
  E.filename = strdup(filename);

  FILE *fp = fopen(filename, "r");
  if (!fp) die("fopen");

  char *line = NULL;
  size_t linecap = 0;
  ssize_t linelen;
  while ((linelen = getline(&line, &linecap, fp)) != -1) {
    while (linelen > 0 && (line[linelen - 1] == '\n' ||
                           line[linelen - 1] == '\r'))
      linelen--;
    editorAppendRow(line, linelen);
  }
  free(line);
  fclose(fp);
  E.dirty = 0;
}

void editorSave() {
  if (E.filename == NULL) return;

  int len;
  char *buf = editorRowsToString(&len);

  int fd = open(E.filename, O_RDWR | O_CREAT, 0644);
  if (fd != -1) {
    if (ftruncate(fd, len) != -1) {
      if (write(fd, buf, len) == len) {
        close(fd);
        free(buf);
        E.dirty = 0;
        editorSetStatusMessage("%d bytes written to disk", len);
        return;
      }
    }
    close(fd);
  }
```

```
  free(buf);
  editorSetStatusMessage("Can't save! I/O error: %s", strerror(errno));
}

/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

⤢ compiles

Now you should see `(modified)` appear in the status bar when you first insert a character, and you should see it disappear when you save the file to disk.

Quit confirmation

Now we're ready to warn the user about unsaved changes when they try to quit. If `E.dirty` is set, we will display a warning in the status bar, and require the user to press `Ctrl-Q` three more times in order to quit without saving.

kilo.c **Step 115** quit-confirmation

```
/*** includes ***/
/*** defines ***/

#define KILO_VERSION "0.0.1"
#define KILO_TAB_STOP 8
#define KILO_QUIT_TIMES 3

#define CTRL_KEY(k) ((k) & 0x1f)

enum editorKey { … };

/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/

void editorMoveCursor(int key) { … }

void editorProcessKeypress() {
  static int quit_times = KILO_QUIT_TIMES;

  int c = editorReadKey();

  switch (c) {
    case '\r':
```

```
    /* TODO */
    break;

  case CTRL_KEY('q'):
    if (E.dirty && quit_times > 0) {
      editorSetStatusMessage("WARNING!!! File has unsaved changes. "
        "Press Ctrl-Q %d more times to quit.", quit_times);
      quit_times--;
      return;
    }
    write(STDOUT_FILENO, "\x1b[2J", 4);
    write(STDOUT_FILENO, "\x1b[H", 3);
    exit(0);
    break;

  case CTRL_KEY('s'):
    editorSave();
    break;

  case HOME_KEY:
    E.cx = 0;
    break;

  case END_KEY:
    if (E.cy < E.numrows)
      E.cx = E.row[E.cy].size;
    break;

  case BACKSPACE:
  case CTRL_KEY('h'):
  case DEL_KEY:
    /* TODO */
    break;

  case PAGE_UP:
  case PAGE_DOWN:
    {
      if (c == PAGE_UP) {
        E.cy = E.rowoff;
      } else if (c == PAGE_DOWN) {
        E.cy = E.rowoff + E.screenrows - 1;
        if (E.cy > E.numrows) E.cy = E.numrows;
      }

      int times = E.screenrows;
      while (times--)
        editorMoveCursor(c == PAGE_UP ? ARROW_UP : ARROW_DOWN);
    }
    break;

  case ARROW_UP:
  case ARROW_DOWN:
  case ARROW_LEFT:
```

```
    case ARROW_RIGHT:
      editorMoveCursor(c);
      break;

    case CTRL_KEY('l'):
    case '\x1b':
      break;

    default:
      editorInsertChar(c);
      break;
  }

  quit_times = KILO_QUIT_TIMES;
}

/*** init ***/
```

⤢ compiles

We use a static variable in `editorProcessKeypress()` to keep track of how many more times the user must press `Ctrl-Q` to quit. Each time they press `Ctrl-Q` with unsaved changes, we set the status message and decrement `quit_times`. When `quit_times` gets to `0`, we finally allow the program to exit. When they press any key other than `Ctrl-Q`, then `quit_times` gets reset back to `3` at the end of the `editorProcessKeypress()` function.

Simple backspacing

Let's implement backspacing next. First we'll create an `editorRowDelChar()` function, which deletes a character in an `erow`.

kilo.c **Step 116** `row-del-char`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/

int editorRowCxToRx(erow *row, int cx) { … }

void editorUpdateRow(erow *row) { … }

void editorAppendRow(char *s, size_t len) { … }

void editorRowInsertChar(erow *row, int at, int c) { … }

void editorRowDelChar(erow *row, int at) {
  if (at < 0 || at >= row->size) return;
```

```
  memmove(&row->chars[at], &row->chars[at + 1], row->size - at);
  row->size--;
  editorUpdateRow(row);
  E.dirty++;
}

/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

compiles, but with no observable effects

As you can see, it's very similar to `editorRowInsertChar()`, except we don't have any memory management to do. We just use `memmove()` to overwrite the deleted character with the characters that come after it (notice that the null byte at the end gets included in the move). Then we decrement the row's `size`, call `editorUpdateRow()`, and increment `E.dirty`.

Now let's implement `editorDelChar()`, which uses `editorRowDelChar()` to delete the character that is to the left of the cursor.

kilo.c **Step 117** editor-del-char

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/

void editorInsertChar(int c) { … }

void editorDelChar() {
  if (E.cy == E.numrows) return;

  erow *row = &E.row[E.cy];
  if (E.cx > 0) {
    editorRowDelChar(row, E.cx - 1);
    E.cx--;
  }
}

/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

compiles, but with no observable effects

If the cursor's past the end of the file, then there is nothing to delete, and we `return` immediately. Otherwise, we get the `erow` the cursor is on, and if there is a character to the left of the cursor, we delete it and move the cursor one to the left.

Let's map the `Backspace`, `Ctrl-H`, and `Delete` keys to `editorDelChar()`.

kilo.c **Step 118** key-del-char

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/

void editorMoveCursor(int key) { … }

void editorProcessKeypress() {
  static int quit_times = KILO_QUIT_TIMES;

  int c = editorReadKey();

  switch (c) {
    case '\r':
      /* TODO */
      break;

    case CTRL_KEY('q'):
      if (E.dirty && quit_times > 0) {
        editorSetStatusMessage("WARNING!!! File has unsaved changes. "
          "Press Ctrl-Q %d more times to quit.", quit_times);
        quit_times--;
        return;
      }
      write(STDOUT_FILENO, "\x1b[2J", 4);
      write(STDOUT_FILENO, "\x1b[H", 3);
      exit(0);
      break;

    case CTRL_KEY('s'):
      editorSave();
      break;

    case HOME_KEY:
      E.cx = 0;
      break;
```

```
    case END_KEY:
      if (E.cy < E.numrows)
        E.cx = E.row[E.cy].size;
      break;

    case BACKSPACE:
    case CTRL_KEY('h'):
    case DEL_KEY:
      if (c == DEL_KEY) editorMoveCursor(ARROW_RIGHT);
      editorDelChar();
      break;

    case PAGE_UP:
    case PAGE_DOWN:
      {
        if (c == PAGE_UP) {
          E.cy = E.rowoff;
        } else if (c == PAGE_DOWN) {
          E.cy = E.rowoff + E.screenrows - 1;
          if (E.cy > E.numrows) E.cy = E.numrows;
        }

        int times = E.screenrows;
        while (times--)
          editorMoveCursor(c == PAGE_UP ? ARROW_UP : ARROW_DOWN);
      }
      break;

    case ARROW_UP:
    case ARROW_DOWN:
    case ARROW_LEFT:
    case ARROW_RIGHT:
      editorMoveCursor(c);
      break;

    case CTRL_KEY('l'):
    case '\x1b':
      break;

    default:
      editorInsertChar(c);
      break;
  }

  quit_times = KILO_QUIT_TIMES;
}

/*** init ***/
```

⤢ compiles

It so happens that in our editor, pressing the `→` key and then `Backspace` is equivalent to what you would expect from pressing the `Delete` key in a text editor: it deletes the

character to the right of the cursor. So that is how we implement the `Delete` key above.

Backspacing at the start of a line

Currently, `editorDelChar()` doesn't do anything when the cursor is at the beginning of a line. When the user backspaces at the beginning of a line, we want to append the contents of that line to the previous line, and then delete the current line. This effectively backspaces the implicit `\n` character in between the two lines to join them into one line.

So we need two new row operations: one for appending a string to a row, and one for deleting a row. Let's start by implementing `editorDelRow()`, which will also require an `editorFreeRow()` function for freeing the memory owned by the `erow` we are deleting.

kilo.c **Step 119** del-row

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/

int editorRowCxToRx(erow *row, int cx) { … }

void editorUpdateRow(erow *row) { … }

void editorAppendRow(char *s, size_t len) { … }

void editorFreeRow(erow *row) {
  free(row->render);
  free(row->chars);
}

void editorDelRow(int at) {
  if (at < 0 || at >= E.numrows) return;
  editorFreeRow(&E.row[at]);
  memmove(&E.row[at], &E.row[at + 1], sizeof(erow) * (E.numrows - at - 1));
  E.numrows--;
  E.dirty++;
}

void editorRowInsertChar(erow *row, int at, int c) { … }

void editorRowDelChar(erow *row, int at) { … }

/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
```

```
/*** output ***/
/*** input ***/
/*** init ***/
```

compiles, but with no observable effects

`editorDelRow()` looks a lot like `editorRowDelChar()`, because in both cases we are deleting a single element from an array of elements by its index.

First we validate the `at` index. Then we free the memory owned by the row using `editorFreeRow()`. We then use `memmove()` to overwrite the deleted row struct with the rest of the rows that come after it, and decrement `numrows`. Finally, we increment `E.dirty`.

Now let's implement `editorRowAppendString()`, which appends a string to the end of a row.

kilo.c **Step 120** `row-append-string`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/

int editorRowCxToRx(erow *row, int cx) { … }

void editorUpdateRow(erow *row) { … }

void editorAppendRow(char *s, size_t len) { … }

void editorFreeRow(erow *row) { … }

void editorDelRow(int at) { … }

void editorRowInsertChar(erow *row, int at, int c) { … }

void editorRowAppendString(erow *row, char *s, size_t len) {
  row->chars = realloc(row->chars, row->size + len + 1);
  memcpy(&row->chars[row->size], s, len);
  row->size += len;
  row->chars[row->size] = '\0';
  editorUpdateRow(row);
  E.dirty++;
}

void editorRowDelChar(erow *row, int at) { … }

/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
```

```
/*** output ***/
/*** input ***/
/*** init ***/
```

compiles, but with no observable effects

The row's new size is `row->size + len + 1` (including the null byte), so first we allocate that much memory for `row->chars`. Then we simply `memcpy()` the given string to the end of the contents of `row->chars`. We update `row->size`, call `editorUpdateRow()` as usual, and increment `E.dirty` as usual.

Now we're ready to get `editorDelChar()` to handle the case where the cursor is at the beginning of a line.

kilo.c **Step 121** del-char-row

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/

void editorInsertChar(int c) { … }

void editorDelChar() {
  if (E.cy == E.numrows) return;
  if (E.cx == 0 && E.cy == 0) return;

  erow *row = &E.row[E.cy];
  if (E.cx > 0) {
    editorRowDelChar(row, E.cx - 1);
    E.cx--;
  } else {
    E.cx = E.row[E.cy - 1].size;
    editorRowAppendString(&E.row[E.cy - 1], row->chars, row->size);
    editorDelRow(E.cy);
    E.cy--;
  }
}

/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

compiles

If the cursor is at the beginning of the *first* line, then there's nothing to do, so we `return` immediately. Otherwise, if we find that `E.cx == 0`, we call `editorRowAppendString()` and then `editorDelRow()` as we planned. `row`

points to the row we are deleting, so we append `row->chars` to the previous row, and then delete the row that `E.cy` is on. We set `E.cx` to the end of the contents of the previous row before appending to that row. That way, the cursor will end up at the point where the two lines joined.

Notice that pressing the `Delete` key at the end of a line works as the user would expect, joining the current line with the next line. This is because moving the cursor to the right at the end of a line moves it to the beginning of the next line. So making the `Delete` key an alias for the `→` key followed by the `Backspace` key still works.

The `Enter` key

The last editor operation we have to implement is the `Enter` key. The `Enter` key allows the user to insert new lines into the text, or split a line into two lines. The first thing we need to do is rename the `editorAppendRow(...)` function to `editorInsertRow(int at, ...)`. It will now be able to insert a row at the index specified by the new `at` argument.

kilo.c **Step 122** append-to-insert

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/

int editorRowCxToRx(erow *row, int cx) { … }

void editorUpdateRow(erow *row) { … }

void editorInsertRow(int at, char *s, size_t len) {
  if (at < 0 || at > E.numrows) return;

  E.row = realloc(E.row, sizeof(erow) * (E.numrows + 1));
  memmove(&E.row[at + 1], &E.row[at], sizeof(erow) * (E.numrows - at));

  int at = E.numrows;
  E.row[at].size = len;
  E.row[at].chars = malloc(len + 1);
  memcpy(E.row[at].chars, s, len);
  E.row[at].chars[len] = '\0';

  E.row[at].rsize = 0;
  E.row[at].render = NULL;
  editorUpdateRow(&E.row[at]);

  E.numrows++;
  E.dirty++;
```

```
}

void editorFreeRow(erow *row) { … }

void editorDelRow(int at) { … }

void editorRowInsertChar(erow *row, int at, int c) { … }

void editorRowAppendString(erow *row, char *s, size_t len) { … }

void editorRowDelChar(erow *row, int at) { … }

/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

℩ doesn't compile

Much like `editorRowInsertChar()`, we first validate `at`, then allocate memory for one more `erow`, and use `memmove()` to make room at the specified index for the new row.

We also delete the old `int at = ...` line, since `at` is now being passed in as an argument.

We now have to replace all calls to `editorAppendRow(...)` with calls to `editorInsertRow(E.numrows, ...)`.

kilo.c **Step 123** `use-insert-row`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/

void editorInsertChar(int c) {
  if (E.cy == E.numrows) {
    editorInsertRow(E.numrows, "", 0);
  }
  editorRowInsertChar(&E.row[E.cy], E.cx, c);
  E.cx++;
}

void editorDelChar() { … }

/*** file i/o ***/
```

```
char *editorRowsToString(int *buflen) { … }

void editorOpen(char *filename) {
  free(E.filename);
  E.filename = strdup(filename);

  FILE *fp = fopen(filename, "r");
  if (!fp) die("fopen");

  char *line = NULL;
  size_t linecap = 0;
  ssize_t linelen;
  while ((linelen = getline(&line, &linecap, fp)) != -1) {
    while (linelen > 0 && (line[linelen - 1] == '\n' ||
                           line[linelen - 1] == '\r'))
      linelen--;
    editorInsertRow(E.numrows, line, linelen);
  }
  free(line);
  fclose(fp);
  E.dirty = 0;
}

void editorSave() { … }

/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

compiles, but with no observable effects

Now that we have `editorInsertRow()`, we're ready to implement `editorInsertNewline()`, which handles an `Enter` keypress.

kilo.c **Step 124** `insert-newline`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/

void editorInsertChar(int c) { … }

void editorInsertNewline() {
  if (E.cx == 0) {
    editorInsertRow(E.cy, "", 0);
  } else {
    erow *row = &E.row[E.cy];
    editorInsertRow(E.cy + 1, &row->chars[E.cx], row->size - E.cx);
    row = &E.row[E.cy];
```

```
    row->size = E.cx;
    row->chars[row->size] = '\0';
    editorUpdateRow(row);
  }
  E.cy++;
  E.cx = 0;
}

void editorDelChar() { … }

/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

compiles, but with no observable effects

If we're at the beginning of a line, all we have to do is insert a new blank row before the line we're on.

Otherwise, we have to split the line we're on into two rows. First we call `editorInsertRow()` and pass it the characters on the current row that are to the right of the cursor. That creates a new row after the current one, with the correct contents. Then we reassign the `row` pointer, because `editorInsertRow()` calls `realloc()`, which might move memory around on us and invalidate the pointer (yikes). Then we truncate the current row's contents by setting its size to the position of the cursor, and we call `editorUpdateRow()` on the truncated row. (`editorInsertRow()` already calls `editorUpdateRow()` for the new row.)

In both cases, we increment `E.cy`, and set `E.cx` to `0` to move the cursor to the beginning of the row.

Finally, let's actually map the `Enter` key to the `editorInsertNewline()` operation.

kilo.c **Step 125** `enter-key`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/

void editorMoveCursor(int key) { … }
```

```
void editorProcessKeypress() {
  static int quit_times = KILO_QUIT_TIMES;

  int c = editorReadKey();

  switch (c) {
    case '\r':
      editorInsertNewline();
      break;

    case CTRL_KEY('q'):
      if (E.dirty && quit_times > 0) {
        editorSetStatusMessage("WARNING!!! File has unsaved changes. "
          "Press Ctrl-Q %d more times to quit.", quit_times);
        quit_times--;
        return;
      }
      write(STDOUT_FILENO, "\x1b[2J", 4);
      write(STDOUT_FILENO, "\x1b[H", 3);
      exit(0);
      break;

    case CTRL_KEY('s'):
      editorSave();
      break;

    case HOME_KEY:
      E.cx = 0;
      break;

    case END_KEY:
      if (E.cy < E.numrows)
        E.cx = E.row[E.cy].size;
      break;

    case BACKSPACE:
    case CTRL_KEY('h'):
    case DEL_KEY:
      if (c == DEL_KEY) editorMoveCursor(ARROW_RIGHT);
      editorDelChar();
      break;

    case PAGE_UP:
    case PAGE_DOWN:
      {
        if (c == PAGE_UP) {
          E.cy = E.rowoff;
        } else if (c == PAGE_DOWN) {
          E.cy = E.rowoff + E.screenrows - 1;
          if (E.cy > E.numrows) E.cy = E.numrows;
        }

        int times = E.screenrows;
```

```
        while (times--)
          editorMoveCursor(c == PAGE_UP ? ARROW_UP : ARROW_DOWN);
      }
      break;

    case ARROW_UP:
    case ARROW_DOWN:
    case ARROW_LEFT:
    case ARROW_RIGHT:
      editorMoveCursor(c);
      break;

    case CTRL_KEY('l'):
    case '\x1b':
      break;

    default:
      editorInsertChar(c);
      break;
  }

  quit_times = KILO_QUIT_TIMES;
}

/*** init ***/
```

⤢ compiles

That concludes all of the text editing operations we are going to implement. If you wish, and if you are brave enough, you may now start using the editor to modify its own code for the rest of the tutorial. If you do, I suggest making regular backups of your work (using `git` or similar) in case you run into bugs in the editor.

Save as...

Currently, when the user runs `./kilo` with no arguments, they get a blank file to edit but have no way of saving. We need a way of prompting the user to input a filename when saving a new file. Let's make an `editorPrompt()` function that displays a prompt in the status bar, and lets the user input a line of text after the prompt.

kilo.c **Step 126** `prompt`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/

void editorSetStatusMessage(const char *fmt, ...);
void editorRefreshScreen();

/*** terminal ***/
```

```
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/

char *editorPrompt(char *prompt) {
  size_t bufsize = 128;
  char *buf = malloc(bufsize);

  size_t buflen = 0;
  buf[0] = '\0';

  while (1) {
    editorSetStatusMessage(prompt, buf);
    editorRefreshScreen();

    int c = editorReadKey();
    if (c == '\r') {
      if (buflen != 0) {
        editorSetStatusMessage("");
        return buf;
      }
    } else if (!iscntrl(c) && c < 128) {
      if (buflen == bufsize - 1) {
        bufsize *= 2;
        buf = realloc(buf, bufsize);
      }
      buf[buflen++] = c;
      buf[buflen] = '\0';
    }
  }
}

void editorMoveCursor(int key) { … }

void editorProcessKeypress() { … }

/*** init ***/
```

compiles, but with no observable effects

The user's input is stored in `buf`, which is a dynamically allocated string that we initalize to the empty string. We then enter an infinite loop that repeatedly sets the status message, refreshes the screen, and waits for a keypress to handle. The `prompt` is expected to be a format string containing a `%s`, which is where the user's input will be displayed.

When the user presses `Enter`, and their input is not empty, the status message is cleared and their input is returned. Otherwise, when they input a printable character, we append it to `buf`. If `buflen` has reached the maximum capacity we allocated

(stored in `bufsize`), then we double `bufsize` and allocate that amount of memory before appending to `buf`. We also make sure that `buf` ends with a `\0` character, because both `editorSetStatusMessage()` and the caller of `editorPrompt()` will use it to know where the string ends.

Notice that we have to make sure the input key isn't one of the special keys in the `editorKey` enum, which have high integer values. To do that, we test whether the input key is in the range of a `char` by making sure it is less than `128`.

Now let's prompt the user for a filename in `editorSave()`, when `E.filename` is `NULL`.

kilo.c **Step 127** save-as

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/

void editorSetStatusMessage(const char *fmt, ...);
void editorRefreshScreen();
char *editorPrompt(char *prompt);

/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/

char *editorRowsToString(int *buflen) { … }

void editorOpen(char *filename) { … }

void editorSave() {
  if (E.filename == NULL) {
    E.filename = editorPrompt("Save as: %s");
  }

  int len;
  char *buf = editorRowsToString(&len);

  int fd = open(E.filename, O_RDWR | O_CREAT, 0644);
  if (fd != -1) {
    if (ftruncate(fd, len) != -1) {
      if (write(fd, buf, len) == len) {
        close(fd);
        free(buf);
        E.dirty = 0;
        editorSetStatusMessage("%d bytes written to disk", len);
        return;
      }
    }
```

```
    close(fd);
  }

  free(buf);
  editorSetStatusMessage("Can't save! I/O error: %s", strerror(errno));
}

/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

⤢ compiles

Great, we now have basic "Save as..." functionality. Next, let's allow the user to press `Escape` to cancel the input prompt.

kilo.c **Step 128** `prompt-escape`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/

char *editorPrompt(char *prompt) {
  size_t bufsize = 128;
  char *buf = malloc(bufsize);

  size_t buflen = 0;
  buf[0] = '\0';

  while (1) {
    editorSetStatusMessage(prompt, buf);
    editorRefreshScreen();

    int c = editorReadKey();
    if (c == '\x1b') {
      editorSetStatusMessage("");
      free(buf);
      return NULL;
    } else if (c == '\r') {
      if (buflen != 0) {
        editorSetStatusMessage("");
        return buf;
      }
    } else if (!iscntrl(c) && c < 128) {
      if (buflen == bufsize - 1) {
```

```
        bufsize *= 2;
        buf = realloc(buf, bufsize);
      }
      buf[buflen++] = c;
      buf[buflen] = '\0';
    }
  }
}

void editorMoveCursor(int key) { … }

void editorProcessKeypress() { … }

/*** init ***/
```

⤢ compiles

When an input prompt is cancelled, we `free()` the `buf` ourselves and return `NULL`. So let's handle a return value of `NULL` in `editorSave()` by aborting the save operation and displaying a "Save aborted" message to the user.

kilo.c **Step 129** `abort-save`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/

char *editorRowsToString(int *buflen) { … }

void editorOpen(char *filename) { … }

void editorSave() {
  if (E.filename == NULL) {
    E.filename = editorPrompt("Save as: %s (ESC to cancel)");
    if (E.filename == NULL) {
      editorSetStatusMessage("Save aborted");
      return;
    }
  }

  int len;
  char *buf = editorRowsToString(&len);

  int fd = open(E.filename, O_RDWR | O_CREAT, 0644);
  if (fd != -1) {
    if (ftruncate(fd, len) != -1) {
      if (write(fd, buf, len) == len) {
        close(fd);
        free(buf);
```

```
        E.dirty = 0;
        editorSetStatusMessage("%d bytes written to disk", len);
        return;
      }
    }
    close(fd);
  }

  free(buf);
  editorSetStatusMessage("Can't save! I/O error: %s", strerror(errno));
}

/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

⤢ compiles

(Note: If you're using **Bash on Windows**, you will have to press `Escape` 3 times to get one `Escape` keypress to register in our program, because the `read()` calls in `editorReadKey()` that look for an escape sequence never time out.)

Now let's allow the user to press `Backspace` (or `Ctrl-H`, or `Delete`) in the input prompt.

kilo.c **Step 130** `prompt-backspace`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** prototypes ***/
/*** terminal ***/
/*** row operations ***/
/*** editor operations ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/

char *editorPrompt(char *prompt) {
  size_t bufsize = 128;
  char *buf = malloc(bufsize);

  size_t buflen = 0;
  buf[0] = '\0';

  while (1) {
    editorSetStatusMessage(prompt, buf);
    editorRefreshScreen();

    int c = editorReadKey();
    if (c == DEL_KEY || c == CTRL_KEY('h') || c == BACKSPACE) {
```

```
      if (buflen != 0) buf[--buflen] = '\0';
    } else if (c == '\x1b') {
      editorSetStatusMessage("");
      free(buf);
      return NULL;
    } else if (c == '\r') {
      if (buflen != 0) {
        editorSetStatusMessage("");
        return buf;
      }
    } else if (!iscntrl(c) && c < 128) {
      if (buflen == bufsize - 1) {
        bufsize *= 2;
        buf = realloc(buf, bufsize);
      }
      buf[buflen++] = c;
      buf[buflen] = '\0';
    }
  }
}

void editorMoveCursor(int key) { … }

void editorProcessKeypress() { … }

/*** init ***/
```

⤢ compiles

In the next chapter, we'll make use of `editorPrompt()` to implement an incremental search feature in our editor.

1.0.0beta11 (changelog)

top of page