A text viewer

A line viewer

Let's create a data type for storing a row of text in our editor.

kilo.c Step 55 erow

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

typedef struct erow {
  int size;
  char *chars;
} erow;

struct editorConfig {
  int cx, cy;
  int screenrows;
  int screencols;
  int numrows;
  erow row;
  struct termios orig_termios;
};

struct editorConfig E;

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

void initEditor() {
  E.cx = 0;
  E.cy = 0;
  E.numrows = 0;

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

int main() { … }
```

compiles, but with no observable effects

`erow` stands for "editor row", and stores a line of text as a pointer to the dynamically-allocated character data and a length. The `typedef` lets us refer to the type as `erow` instead of `struct erow`.

We add an `erow` value to the editor global state, as well as a `numrows` variable. For now, the editor will only display a single line of text, and so `numrows` can be either `0` or `1`. We initialize it to `0` in `initEditor()`.

Let's fill that `erow` with some text now. We won't worry about reading from a file just yet. Instead, we'll hardcode a "Hello, world" string into it.

kilo.c **Step 56** hello-world

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

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

/*** defines ***/
/*** data ***/
/*** terminal ***/

void die(const char *s) { … }

void disableRawMode() { … }

void enableRawMode() { … }

int editorReadKey() { … }

int getCursorPosition(int *rows, int *cols) { … }

int getWindowSize(int *rows, int *cols) { … }

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

void editorOpen() {
  char *line = "Hello, world!";
  ssize_t linelen = 13;
```

```
  E.row.size = linelen;
  E.row.chars = malloc(linelen + 1);
  memcpy(E.row.chars, line, linelen);
  E.row.chars[linelen] = '\0';
  E.numrows = 1;
}

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

void initEditor() { … }

int main() {
  enableRawMode();
  initEditor();
  editorOpen();

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

  return 0;
}
```

compiles, but with no observable effects

`malloc()` comes from `<stdlib.h>`. `ssize_t` comes from `<sys/types.h>`.

`editorOpen()` will eventually be for opening and reading a file from disk, so we put it in a new `/*** file i/o ***/` section. To load our "Hello, world" message into the editor's `erow` struct, we set the `size` field to the length of our message, `malloc()` the necessary memory, and `memcpy()` the message to the `chars` field which points to the memory we allocated. Finally, we set the `E.numrows` variable to `1`, to indicate that the `erow` now contains a line that should be displayed.

Let's display it then.

kilo.c **Step 57** `draw-erow`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** file i/o ***/
/*** append buffer ***/
```

```
/*** output ***/

void editorDrawRows(struct abuf *ab) {
  int y;
  for (y = 0; y < E.screenrows; y++) {
    if (y >= E.numrows) {
      if (y == E.screenrows / 3) {
        char welcome[80];
        int welcomelen = snprintf(welcome, sizeof(welcome),
          "Kilo editor -- version %s", KILO_VERSION);
        if (welcomelen > E.screencols) welcomelen = E.screencols;
        int padding = (E.screencols - welcomelen) / 2;
        if (padding) {
          abAppend(ab, "~", 1);
          padding--;
        }
        while (padding--) abAppend(ab, " ", 1);
        abAppend(ab, welcome, welcomelen);
      } else {
        abAppend(ab, "~", 1);
      }
    } else {
      int len = E.row.size;
      if (len > E.screencols) len = E.screencols;
      abAppend(ab, E.row.chars, len);
    }

    abAppend(ab, "\x1b[K", 3);
    if (y < E.screenrows - 1) {
      abAppend(ab, "\r\n", 2);
    }
  }
}

void editorRefreshScreen() { … }

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

↗ compiles

We wrap our previous row-drawing code in an `if` statement that checks whether we are currently drawing a row that is part of the text buffer, or a row that comes after the end of the text buffer.

To draw a row that's part of the text buffer, we simply write out the `chars` field of the `erow`. But first, we take care to truncate the rendered line if it would go past the end of the screen.

Next, let's allow the user to open an actual file. We'll read and display the first line of the file.

kilo.c — Step 58 — open-file

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** file i/o ***/

void editorOpen(char *filename) {
  FILE *fp = fopen(filename, "r");
  if (!fp) die("fopen");

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

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

void initEditor() { … }

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

  while (1) {
    editorRefreshScreen();
```

```
    editorProcessKeypress();
  }

  return 0;
}
```

may or may not compile

`FILE`, `fopen()`, and `getline()` come from `<stdio.h>`.

The core of `editorOpen()` is the same, we just get the `line` and `linelen` values from `getline()` now, instead of hardcoded values.

`editorOpen()` now takes a filename and opens the file for reading using `fopen()`. We allow the user to choose a file to open by checking if they passed a filename as a command line argument. If they did, we call `editorOpen()` and pass it the filename. If they ran `./kilo` with no arguments, `editorOpen()` will not be called and they'll start with a blank file.

`getline()` is useful for reading lines from a file when we don't know how much memory to allocate for each line. It takes care of memory management for you. First, we pass it a null `line` pointer and a `linecap` (line capacity) of `0`. That makes it allocate new memory for the next line it reads, and set `line` to point to the memory, and set `linecap` to let you know how much memory it allocated. Its return value is the length of the line it read, or `-1` if it's at the end of the file and there are no more lines to read. Later, when we have `editorOpen()` read multiple lines of a file, we will be able to feed the new `line` and `linecap` values back into `getline()` over and over, and it will try and reuse the memory that `line` points to as long as the `linecap` is big enough to fit the next line it reads. For now, we just copy the one line it reads into `E.row.chars`, and then `free()` the `line` that `getline()` allocated.

We also strip off the newline or carriage return at the end of the line before copying it into our `erow`. We know each `erow` represents one line of text, so there's no use storing a newline character at the end of each one.

If your compiler complains about `getline()`, you may need to define a feature test macro. Even if it compiles fine on your machine without them, let's add them to make our code more portable.

kilo.c **Step 59** `feature-test-macros`

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

#define _DEFAULT_SOURCE
```

```
#define _BSD_SOURCE
#define _GNU_SOURCE

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

/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/
/*** init ***/
```

↗ compiles

We add them above our includes, because the header files we're including use the macros to decide what features to expose.

Now let's fix a quick bug. We want the welcome message to only display when the user starts the program with no arguments, and not when they open a file, as the welcome message could get in the way of displaying the file.

kilo.c — **Step 60** — hide-welcome

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/

void editorDrawRows(struct abuf *ab) {
  int y;
  for (y = 0; y < E.screenrows; y++) {
    if (y >= E.numrows) {
      if (E.numrows == 0 && y == E.screenrows / 3) {
        char welcome[80];
        int welcomelen = snprintf(welcome, sizeof(welcome),
          "Kilo editor -- version %s", KILO_VERSION);
```

```
        if (welcomelen > E.screencols) welcomelen = E.screencols;
        int padding = (E.screencols - welcomelen) / 2;
        if (padding) {
          abAppend(ab, "~", 1);
          padding--;
        }
        while (padding--) abAppend(ab, " ", 1);
        abAppend(ab, welcome, welcomelen);
      } else {
        abAppend(ab, "~", 1);
      }
    } else {
      int len = E.row.size;
      if (len > E.screencols) len = E.screencols;
      abAppend(ab, E.row.chars, len);
    }

    abAppend(ab, "\x1b[K", 3);
    if (y < E.screenrows - 1) {
      abAppend(ab, "\r\n", 2);
    }
  }
}

void editorRefreshScreen() { … }

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

↗ compiles

There, now the welcome message only displays if the text buffer is completely empty.

Multiple lines

To store multiple lines, let's make `E.row` an array of `erow` structs. It will be a dynamically-allocated array, so we'll make it a pointer to `erow`, and initialize the pointer to `NULL`. (This will break a bunch of our code that doesn't expect `E.row` to be a pointer, so the program will fail to compile for the next few steps.)

kilo.c — Step 61 — erow-array

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

typedef struct erow { … } erow;
```

```
struct editorConfig {
  int cx, cy;
  int screenrows;
  int screencols;
  int numrows;
  erow *row;
  struct termios orig_termios;
};

struct editorConfig E;

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

void initEditor() {
  E.cx = 0;
  E.cy = 0;
  E.numrows = 0;
  E.row = NULL;

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

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

ℳ doesn't compile

Next, let's move the code in `editorOpen()` that initializes `E.row` to a new function called `editorAppendRow()`. We'll also put it under a new section, `/*** row operations ***/`.

kilo.c **Step 62** `append-row`

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

void die(const char *s) { … }

void disableRawMode() { … }

void enableRawMode() { … }

int editorReadKey() { … }
```

```
int getCursorPosition(int *rows, int *cols) { … }

int getWindowSize(int *rows, int *cols) { … }

/*** row operations ***/

void editorAppendRow(char *s, size_t len) {
  E.row.size = len;
  E.row.chars = malloc(len + 1);
  memcpy(E.row.chars, s, len);
  E.row.chars[len] = '\0';
  E.numrows = 1;
}

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

void editorOpen(char *filename) {
  FILE *fp = fopen(filename, "r");
  if (!fp) die("fopen");

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

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

ℳ doesn't compile

Notice that we renamed the `line` and `linelen` variables to `s` and `len`, which are now arguments to `editorAppendRow()`.

We want `editorAppendRow()` to allocate space for a new `erow`, and then copy the given string to a new `erow` at the end of the `E.row` array. Let's do that now.

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

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.numrows++;
}

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

ℳ doesn't compile

We have to tell `realloc()` how many bytes we want to allocate, so we multiply the number of bytes each `erow` takes (`sizeof(erow)`) and multiply that by the number of rows we want. Then we set `at` to the index of the new row we want to initialize, and replace each occurrence of `E.row` with `E.row[at]`. Lastly, we change `E.numrows = 1` to `E.numrows++`.

Next, let's update `editorDrawRows()` to use `E.row[y]` instead of `E.row`, when printing out the current line.

kilo.c **Step 64** `draw-multiple-erows`

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

void editorDrawRows(struct abuf *ab) {
  int y;
  for (y = 0; y < E.screenrows; y++) {
```

```
    if (y >= E.numrows) {
      if (E.numrows == 0 && y == E.screenrows / 3) {
        char welcome[80];
        int welcomelen = snprintf(welcome, sizeof(welcome),
          "Kilo editor -- version %s", KILO_VERSION);
        if (welcomelen > E.screencols) welcomelen = E.screencols;
        int padding = (E.screencols - welcomelen) / 2;
        if (padding) {
          abAppend(ab, "~", 1);
          padding--;
        }
        while (padding--) abAppend(ab, " ", 1);
        abAppend(ab, welcome, welcomelen);
      } else {
        abAppend(ab, "~", 1);
      }
    } else {
      int len = E.row[y].size;
      if (len > E.screencols) len = E.screencols;
      abAppend(ab, E.row[y].chars, len);
    }

    abAppend(ab, "\x1b[K", 3);
    if (y < E.screenrows - 1) {
      abAppend(ab, "\r\n", 2);
    }
  }
}

void editorRefreshScreen() { … }

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

compiles, but with no observable effects

At this point the code should compile, but it still only reads a single line from the file. Let's add a `while` loop to `editorOpen()` to read an entire file into `E.row`.

kilo.c **Step 65** `read-multiple-lines`

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

void editorOpen(char *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);
}

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

↗ compiles

The `while` loop works because `getline()` returns `-1` when it gets to the end of the file and there are no more lines to read.

Now you should see your screen fill up with lines of text when you run `./kilo kilo.c`, for example.

Vertical scrolling

Next we want to enable the user to scroll through the whole file, instead of just being able to see the top few lines of the file. Let's add a `rowoff` (row offset) variable to the global editor state, which will keep track of what row of the file the user is currently scrolled to.

kilo.c — Step 66 — rowoff

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

typedef struct erow { … } erow;

struct editorConfig {
  int cx, cy;
  int rowoff;
  int screenrows;
```

```
  int screencols;
  int numrows;
  erow *row;
  struct termios orig_termios;
};

struct editorConfig E;

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

void initEditor() {
  E.cx = 0;
  E.cy = 0;
  E.rowoff = 0;
  E.numrows = 0;
  E.row = NULL;

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

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

compiles, but with no observable effects

We initialize it to `0`, which means we'll be scrolled to the top of the file by default.

Now let's have `editorDrawRows()` display the correct range of lines of the file according to the value of `rowoff`.

kilo.c — Step 67 — filerow

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

void editorDrawRows(struct abuf *ab) {
  int y;
  for (y = 0; y < E.screenrows; y++) {
```

```
    int filerow = y + E.rowoff;
    if (filerow >= E.numrows) {
      if (E.numrows == 0 && y == E.screenrows / 3) {
        char welcome[80];
        int welcomelen = snprintf(welcome, sizeof(welcome),
          "Kilo editor -- version %s", KILO_VERSION);
        if (welcomelen > E.screencols) welcomelen = E.screencols;
        int padding = (E.screencols - welcomelen) / 2;
        if (padding) {
          abAppend(ab, "~", 1);
          padding--;
        }
        while (padding--) abAppend(ab, " ", 1);
        abAppend(ab, welcome, welcomelen);
      } else {
        abAppend(ab, "~", 1);
      }
    } else {
      int len = E.row[filerow].size;
      if (len > E.screencols) len = E.screencols;
      abAppend(ab, E.row[filerow].chars, len);
    }

    abAppend(ab, "\x1b[K", 3);
    if (y < E.screenrows - 1) {
      abAppend(ab, "\r\n", 2);
    }
  }
}

void editorRefreshScreen() { … }

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

compiles, but with no observable effects

To get the row of the file that we want to display at each `y` position, we add `E.rowoff` to the `y` position. So we define a new variable `filerow` that contains that value, and use that as the index into `E.row`.

Now where do we set the value of `E.rowoff`? Our strategy will be to check if the cursor has moved outside of the visible window, and if so, adjust `E.rowoff` so that the cursor is just inside the visible window. We'll put this logic in a function called `editorScroll()`, and call it right before we refresh the screen.

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

void editorScroll() {
  if (E.cy < E.rowoff) {
    E.rowoff = E.cy;
  }
  if (E.cy >= E.rowoff + E.screenrows) {
    E.rowoff = E.cy - E.screenrows + 1;
  }
}

void editorDrawRows(struct abuf *ab) { … }

void editorRefreshScreen() {
  editorScroll();

  struct abuf ab = ABUF_INIT;

  abAppend(&ab, "\x1b[?25l", 6);
  abAppend(&ab, "\x1b[H", 3);

  editorDrawRows(&ab);

  char buf[32];
  snprintf(buf, sizeof(buf), "\x1b[%d;%dH", E.cy + 1, E.cx + 1);
  abAppend(&ab, buf, strlen(buf));

  abAppend(&ab, "\x1b[?25h", 6);

  write(STDOUT_FILENO, ab.b, ab.len);
  abFree(&ab);
}

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

compiles, but with no observable effects

The first `if` statement checks if the cursor is above the visible window, and if so, scrolls up to where the cursor is. The second `if` statement checks if the cursor is past the bottom of the visible window, and contains slightly more complicated arithmetic

because `E.rowoff` refers to what's at the *top* of the screen, and we have to get `E.screenrows` involved to talk about what's at the *bottom* of the screen.

Now let's allow the cursor to advance past the bottom of the screen (but not past the bottom of the file).

kilo.c — Step 69 — enable-vertical-scroll

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

void editorMoveCursor(int key) {
  switch (key) {
    case ARROW_LEFT:
      if (E.cx != 0) {
        E.cx--;
      }
      break;
    case ARROW_RIGHT:
      if (E.cx != E.screencols - 1) {
        E.cx++;
      }
      break;
    case ARROW_UP:
      if (E.cy != 0) {
        E.cy--;
      }
      break;
    case ARROW_DOWN:
      if (E.cy < E.numrows) {
        E.cy++;
      }
      break;
  }
}

void editorProcessKeypress() { … }

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

↗ compiles

You should be able to scroll through the entire file now, when you run `./kilo kilo.c`. (If the file contains tab characters, you'll see that the characters that the tabs take up aren't being erased properly when drawing to the screen. We'll fix this issue soon. In the meantime, you may want to test with a file that doesn't contain a lot of tabs.)

If you try to scroll back up, you may notice the cursor isn't being positioned properly. That is because `E.cy` no longer refers to the position of the cursor on the screen. It refers to the position of the cursor within the text file. To position the cursor on the screen, we now have to subtract `E.rowoff` from the value of `E.cy`.

kilo.c — Step 70 — fix-cursor-scrolling

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

void editorScroll() { … }

void editorDrawRows(struct abuf *ab) { … }

void editorRefreshScreen() {
  editorScroll();

  struct abuf ab = ABUF_INIT;

  abAppend(&ab, "\x1b[?25l", 6);
  abAppend(&ab, "\x1b[H", 3);

  editorDrawRows(&ab);

  char buf[32];
  snprintf(buf, sizeof(buf), "\x1b[%d;%dH", (E.cy - E.rowoff) + 1, E.cx + 1);
  abAppend(&ab, buf, strlen(buf));

  abAppend(&ab, "\x1b[?25h", 6);

  write(STDOUT_FILENO, ab.b, ab.len);
  abFree(&ab);
}

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

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

↗ compiles

Horizontal scrolling

Now let's work on horizontal scrolling. We'll implement it in just about the same way we implemented vertical scrolling. Start by adding a `coloff` (column offset) variable to the global editor state.

kilo.c — Step 71 — coloff

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

typedef struct erow { … } erow;

struct editorConfig {
  int cx, cy;
  int rowoff;
  int coloff;
  int screenrows;
  int screencols;
  int numrows;
  erow *row;
  struct termios orig_termios;
};

struct editorConfig E;

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

void initEditor() {
  E.cx = 0;
  E.cy = 0;
  E.rowoff = 0;
  E.coloff = 0;
  E.numrows = 0;
  E.row = NULL;

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

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

compiles, but with no observable effects

To display each row at the column offset, we'll use `E.coloff` as an index into the `chars` of each `erow` we display, and subtract the number of characters that are to the left of the offset from the length of the row.

kilo.c — Step 72 — use-coloff

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

void editorScroll() { … }

void editorDrawRows(struct abuf *ab) {
  int y;
  for (y = 0; y < E.screenrows; y++) {
    int filerow = y + E.rowoff;
    if (filerow >= E.numrows) {
      if (E.numrows == 0 && y == E.screenrows / 3) {
        char welcome[80];
        int welcomelen = snprintf(welcome, sizeof(welcome),
          "Kilo editor -- version %s", KILO_VERSION);
        if (welcomelen > E.screencols) welcomelen = E.screencols;
        int padding = (E.screencols - welcomelen) / 2;
        if (padding) {
          abAppend(ab, "~", 1);
          padding--;
        }
        while (padding--) abAppend(ab, " ", 1);
        abAppend(ab, welcome, welcomelen);
      } else {
        abAppend(ab, "~", 1);
      }
    } else {
      int len = E.row[filerow].size - E.coloff;
      if (len < 0) len = 0;
      if (len > E.screencols) len = E.screencols;
      abAppend(ab, &E.row[filerow].chars[E.coloff], len);
    }
```

```
    abAppend(ab, "\x1b[K", 3);
    if (y < E.screenrows - 1) {
      abAppend(ab, "\r\n", 2);
    }
  }
}

void editorRefreshScreen() { … }

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

compiles, but with no observable effects

Note that when subtracting `E.coloff` from the length, `len` can now be a negative number, meaning the user scrolled horizontally past the end of the line. In that case, we set `len` to `0` so that nothing is displayed on that line.

Now let's update `editorScroll()` to handle horizontal scrolling.

kilo.c **Step 73** `editor-scroll-horizontal`

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

void editorScroll() {
  if (E.cy < E.rowoff) {
    E.rowoff = E.cy;
  }
  if (E.cy >= E.rowoff + E.screenrows) {
    E.rowoff = E.cy - E.screenrows + 1;
  }
  if (E.cx < E.coloff) {
    E.coloff = E.cx;
  }
  if (E.cx >= E.coloff + E.screencols) {
    E.coloff = E.cx - E.screencols + 1;
  }
}

void editorDrawRows(struct abuf *ab) { … }

void editorRefreshScreen() { … }
```

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

⚖ compiles, but with no observable effects

As you can see, it is exactly parallel to the vertical scrolling code. We just replace `E.cy` with `E.cx`, `E.rowoff` with `E.coloff`, and `E.screenrows` with `E.screencols`.

Now let's allow the user to scroll past the right edge of the screen.

kilo.c — Step 74 — enable-horizontal-scroll

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

void editorMoveCursor(int key) {
  switch (key) {
    case ARROW_LEFT:
      if (E.cx != 0) {
        E.cx--;
      }
      break;
    case ARROW_RIGHT:
      if (E.cx != E.screencols - 1) {
      E.cx++;
      }
      break;
    case ARROW_UP:
      if (E.cy != 0) {
        E.cy--;
      }
      break;
    case ARROW_DOWN:
      if (E.cy < E.numrows) {
        E.cy++;
      }
      break;
  }
}

void editorProcessKeypress() { … }
```

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

↗ compiles

You should be able to confirm that horizontal scrolling now works.

Next, let's fix the cursor positioning, just like we did with vertical scrolling.

kilo.c — Step 75 — fix-cursor-scrolling-horizontal

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

void editorScroll() { … }

void editorDrawRows(struct abuf *ab) { … }

void editorRefreshScreen() {
  editorScroll();

  struct abuf ab = ABUF_INIT;

  abAppend(&ab, "\x1b[?25l", 6);
  abAppend(&ab, "\x1b[H", 3);

  editorDrawRows(&ab);

  char buf[32];
  snprintf(buf, sizeof(buf), "\x1b[%d;%dH", (E.cy - E.rowoff) + 1,
                                            (E.cx - E.coloff) + 1);
  abAppend(&ab, buf, strlen(buf));

  abAppend(&ab, "\x1b[?25h", 6);

  write(STDOUT_FILENO, ab.b, ab.len);
  abFree(&ab);
}

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

↗ compiles

Limit scrolling to the right

Now both `E.cx` and `E.cy` refer to the cursor's position within the file, not its position on the screen. So our goal with the next few steps is to limit the values of `E.cx` and `E.cy` to only ever point to valid positions in the file. Otherwise, the user could move the cursor way off to the right of a line and start inserting text there, which wouldn't make much sense. (The only exceptions to this rule are that `E.cx` can point one character past the end of a line so that characters can be inserted at the end of the line, and `E.cy` can point one line past the end of the file so that new lines at the end of the file can be added easily.)

Let's start by not allowing the user to scroll past the end of the current line.

kilo.c — Step 76 — scroll-limits

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

void editorMoveCursor(int key) {
  erow *row = (E.cy >= E.numrows) ? NULL : &E.row[E.cy];

  switch (key) {
    case ARROW_LEFT:
      if (E.cx != 0) {
        E.cx--;
      }
      break;
    case ARROW_RIGHT:
      if (row && E.cx < row->size) {
        E.cx++;
      }
      break;
    case ARROW_UP:
      if (E.cy != 0) {
        E.cy--;
      }
      break;
    case ARROW_DOWN:
      if (E.cy < E.numrows) {
        E.cy++;
      }
      break;
```

```
  }
}

void editorProcessKeypress() { … }

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

↗ compiles

Since `E.cy` is allowed to be one past the last line of the file, we use the ternary operator to check if the cursor is on an actual line. If it is, then the `row` variable will point to the `erow` that the cursor is on, and we'll check whether `E.cx` is to the left of the end of that line before we allow the cursor to move to the right.

Snap cursor to end of line

The user is still able to move the cursor past the end of a line, however. They can do it by moving the cursor to the end of a long line, then moving it down to the next line, which is shorter. The `E.cx` value won't change, and the cursor will be off to the right of the end of the line it's now on.

Let's add some code to `editorMoveCursor()` that corrects `E.cx` if it ends up past the end of the line it's on.

kilo.c — Step 77 — snap-cursor

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

void editorMoveCursor(int key) {
  erow *row = (E.cy >= E.numrows) ? NULL : &E.row[E.cy];

  switch (key) {
    case ARROW_LEFT:
      if (E.cx != 0) {
        E.cx--;
      }
      break;
    case ARROW_RIGHT:
      if (row && E.cx < row->size) {
```

```
        E.cx++;
      }
      break;
    case ARROW_UP:
      if (E.cy != 0) {
        E.cy--;
      }
      break;
    case ARROW_DOWN:
      if (E.cy < E.numrows) {
        E.cy++;
      }
      break;
  }

  row = (E.cy >= E.numrows) ? NULL : &E.row[E.cy];
  int rowlen = row ? row->size : 0;
  if (E.cx > rowlen) {
    E.cx = rowlen;
  }
}

void editorProcessKeypress() { … }

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

↗ compiles

We have to set `row` again, since `E.cy` could point to a different line than it did before. We then set `E.cx` to the end of that line if `E.cx` is to the right of the end of that line. Also note that we consider a `NULL` line to be of length `0`, which works for our purposes here.

Moving left at the start of a line

Let's allow the user to press ← at the beginning of the line to move to the end of the previous line.

kilo.c **Step 78** `moving-left`

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

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

void editorMoveCursor(int key) {
  erow *row = (E.cy >= E.numrows) ? NULL : &E.row[E.cy];

  switch (key) {
    case ARROW_LEFT:
      if (E.cx != 0) {
        E.cx--;
      } else if (E.cy > 0) {
        E.cy--;
        E.cx = E.row[E.cy].size;
      }
      break;
    case ARROW_RIGHT:
      if (row && E.cx < row->size) {
        E.cx++;
      }
      break;
    case ARROW_UP:
      if (E.cy != 0) {
        E.cy--;
      }
      break;
    case ARROW_DOWN:
      if (E.cy < E.numrows) {
        E.cy++;
      }
      break;
  }

  row = (E.cy >= E.numrows) ? NULL : &E.row[E.cy];
  int rowlen = row ? row->size : 0;
  if (E.cx > rowlen) {
    E.cx = rowlen;
  }
}

void editorProcessKeypress() { … }

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

↗ compiles

We make sure they aren't on the very first line before we move them up a line.

Moving right at the end of a line

Similarly, let's allow the user to press → at the end of a line to go to the beginning of the next line.

kilo.c — Step 79 — moving-right

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

void editorMoveCursor(int key) {
  erow *row = (E.cy >= E.numrows) ? NULL : &E.row[E.cy];

  switch (key) {
    case ARROW_LEFT:
      if (E.cx != 0) {
        E.cx--;
      } else if (E.cy > 0) {
        E.cy--;
        E.cx = E.row[E.cy].size;
      }
      break;
    case ARROW_RIGHT:
      if (row && E.cx < row->size) {
        E.cx++;
      } else if (row && E.cx == row->size) {
        E.cy++;
        E.cx = 0;
      }
      break;
    case ARROW_UP:
      if (E.cy != 0) {
        E.cy--;
      }
      break;
    case ARROW_DOWN:
      if (E.cy < E.numrows) {
        E.cy++;
      }
      break;
  }

  row = (E.cy >= E.numrows) ? NULL : &E.row[E.cy];
  int rowlen = row ? row->size : 0;
```

```
  if (E.cx > rowlen) {
    E.cx = rowlen;
  }
}

void editorProcessKeypress() { … }

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

↗ compiles

Here we have to make sure they're not at the end of the file before moving down a line.

Rendering tabs

If you try opening the `Makefile` using `./kilo Makefile`, you'll notice that the tab character on the second line of the Makefile takes up a width of 8 columns or so. The length of a tab is up to the terminal being used and its settings. We want to *know* the length of each tab, and we also want control over how to render tabs, so we're going to add a second string to the `erow` struct called `render`, which will contain the actual characters to draw on the screen for that row of text. We'll only use `render` for tabs for now, but in the future it could be used to render nonprintable control characters as a `^` character followed by another character, such as `^A` for the `Ctrl-A` character (this is a common way to display control characters in the terminal).

You may also notice that when the tab character in the `Makefile` is displayed by the terminal, it doesn't erase any characters on the screen within that tab. All a tab does is move the cursor forward to the next tab stop, similar to a carriage return or newline. This is another reason why we want to render tabs as multiple spaces, since spaces erase whatever character was there before.

So, let's start by adding `render` and `rsize` (which contains the size of the contents of `render`) to the `erow` struct, and initializing them in `editorAppendRow()`, which is where new `erow`s get constructed and initialized.

kilo.c — Step 80 — render

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

typedef struct erow {
  int size;
  int rsize;
  char *chars;
```

```
  char *render;
} erow;

struct editorConfig { … };

struct editorConfig E;

/*** terminal ***/
/*** row operations ***/

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;

  E.numrows++;
}

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

compiles, but with no observable effects

Next, let's make an `editorUpdateRow()` function that uses the `chars` string of an `erow` to fill in the contents of the `render` string. We'll copy each character from `chars` to `render`. We won't worry about how to render tabs just yet.

kilo.c — Step 81 — editor-update-row

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

void editorUpdateRow(erow *row) {
  free(row->render);
  row->render = malloc(row->size + 1);
```

```
  int j;
  int idx = 0;
  for (j = 0; j < row->size; j++) {
    row->render[idx++] = row->chars[j];
  }
  row->render[idx] = '\0';
  row->rsize = idx;
}

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++;
}

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

compiles, but with no observable effects

After the `for` loop, `idx` contains the number of characters we copied into `row->render`, so we assign it to `row->rsize`.

Now let's replace `chars` and `size` with `render` and `rsize` in `editorDrawRows()`, when we display each `erow`.

kilo.c **Step 82** `use-render`

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

```
void editorScroll() { … }

void editorDrawRows(struct abuf *ab) {
  int y;
  for (y = 0; y < E.screenrows; y++) {
    int filerow = y + E.rowoff;
    if (filerow >= E.numrows) {
      if (E.numrows == 0 && y == E.screenrows / 3) {
        char welcome[80];
        int welcomelen = snprintf(welcome, sizeof(welcome),
          "Kilo editor -- version %s", KILO_VERSION);
        if (welcomelen > E.screencols) welcomelen = E.screencols;
        int padding = (E.screencols - welcomelen) / 2;
        if (padding) {
          abAppend(ab, "~", 1);
          padding--;
        }
        while (padding--) abAppend(ab, " ", 1);
        abAppend(ab, welcome, welcomelen);
      } else {
        abAppend(ab, "~", 1);
      }
    } else {
      int len = E.row[filerow].rsize - E.coloff;
      if (len < 0) len = 0;
      if (len > E.screencols) len = E.screencols;
      abAppend(ab, &E.row[filerow].render[E.coloff], len);
    }

    abAppend(ab, "\x1b[K", 3);
    if (y < E.screenrows - 1) {
      abAppend(ab, "\r\n", 2);
    }
  }
}

void editorRefreshScreen() { … }

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

compiles, but with no observable effects

Now the text viewer is displaying the characters in `render`. Let's add code to `editorUpdateRow()` that renders tabs as multiple space characters.

kilo.c **Step 83** tabs

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

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

void editorUpdateRow(erow *row) {
  int tabs = 0;
  int j;
  for (j = 0; j < row->size; j++)
    if (row->chars[j] == '\t') tabs++;

  free(row->render);
  row->render = malloc(row->size + tabs*7 + 1);

  int idx = 0;
  for (j = 0; j < row->size; j++) {
    if (row->chars[j] == '\t') {
      row->render[idx++] = ' ';
      while (idx % 8 != 0) row->render[idx++] = ' ';
    } else {
      row->render[idx++] = row->chars[j];
    }
  }
  row->render[idx] = '\0';
  row->rsize = idx;
}

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

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

↗ compiles

First, we have to loop through the `chars` of the row and count the tabs in order to know how much memory to allocate for `render`. The maximum number of characters needed for each tab is 8. `row->size` already counts 1 for each tab, so we multiply the number of tabs by 7 and add that to `row->size` to get the maximum amount of memory we'll need for the rendered row.

After allocating the memory, we modify the `for` loop to check whether the current character is a tab. If it is, we append one space (because each tab must advance the cursor forward at least one column), and then append spaces until we get to a tab stop, which is a column that is divisible by 8.

At this point, we should probably make the length of a tab stop a constant.

kilo.c — Step 84 — tab-stop

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

#define KILO_VERSION "0.0.1"
#define KILO_TAB_STOP 8

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

enum editorKey { … };

/*** data ***/
/*** terminal ***/
/*** row operations ***/

void editorUpdateRow(erow *row) {
  int tabs = 0;
  int j;
  for (j = 0; j < row->size; j++)
    if (row->chars[j] == '\t') tabs++;

  free(row->render);
  row->render = malloc(row->size + tabs*(KILO_TAB_STOP - 1) + 1);

  int idx = 0;
  for (j = 0; j < row->size; j++) {
    if (row->chars[j] == '\t') {
      row->render[idx++] = ' ';
      while (idx % KILO_TAB_STOP != 0) row->render[idx++] = ' ';
    } else {
      row->render[idx++] = row->chars[j];
    }
  }
  row->render[idx] = '\0';
  row->rsize = idx;
}

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

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

compiles, but with no observable effects

This makes the code clearer, and also makes the tab stop length configurable.

Tabs and the cursor

The cursor doesn't currently interact with tabs very well. When we position the cursor on the screen, we're still assuming each character takes up only one column on the screen. To fix this, let's introduce a new horizontal coordinate variable, `E.rx`. While `E.cx` is an index into the `chars` field of an `erow`, the `E.rx` variable will be an index into the `render` field. If there are no tabs on the current line, then `E.rx` will be the same as `E.cx`. If there are tabs, then `E.rx` will be greater than `E.cx` by however many extra spaces those tabs take up when rendered.

Start by adding `rx` to the global state struct, and initializing it to `0`.

kilo.c — Step 85 — rx

```
/*** 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;
  struct termios orig_termios;
};

struct editorConfig E;

/*** terminal ***/
/*** row 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;

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

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

compiles, but with no observable effects

We'll set the value of `E.rx` at the top of `editorScroll()`. For now we'll just set it to be the same as `E.cx`. Then we'll replace all instances of `E.cx` with `E.rx` in `editorScroll()`, because scrolling should take into account the characters that are actually rendered to the screen, and the rendered position of the cursor.

kilo.c **Step 86** `rx-scroll`

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

void editorScroll() {
  E.rx = E.cx;

  if (E.cy < E.rowoff) {
    E.rowoff = E.cy;
  }
  if (E.cy >= E.rowoff + E.screenrows) {
    E.rowoff = E.cy - E.screenrows + 1;
  }
  if (E.rx < E.coloff) {
    E.coloff = E.rx;
  }
  if (E.rx >= E.coloff + E.screencols) {
    E.coloff = E.rx - E.screencols + 1;
  }
}

void editorDrawRows(struct abuf *ab) { … }
```

```
void editorRefreshScreen() { … }

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

compiles, but with no observable effects

Now change `E.cx` to `E.rx` in `editorRefreshScreen()` where we set the cursor position.

kilo.c **Step 87** use-rx

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

void editorScroll() { … }

void editorDrawRows(struct abuf *ab) { … }

void editorRefreshScreen() {
  editorScroll();

  struct abuf ab = ABUF_INIT;

  abAppend(&ab, "\x1b[?25l", 6);
  abAppend(&ab, "\x1b[H", 3);

  editorDrawRows(&ab);

  char buf[32];
  snprintf(buf, sizeof(buf), "\x1b[%d;%dH", (E.cy - E.rowoff) + 1,
                                            (E.rx - E.coloff) + 1);
  abAppend(&ab, buf, strlen(buf));

  abAppend(&ab, "\x1b[?25h", 6);

  write(STDOUT_FILENO, ab.b, ab.len);
  abFree(&ab);
}

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

compiles, but with no observable effects

All that's left to do is calculate the value of `E.rx` properly in `editorScroll()`. Let's create an `editorRowCxToRx()` function that converts a `chars` index into a `render` index. We'll need to loop through all the characters to the left of `cx`, and figure out how many spaces each tab takes up.

kilo.c — Step 88 — cx-to-rx

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

int editorRowCxToRx(erow *row, int cx) {
  int rx = 0;
  int j;
  for (j = 0; j < cx; j++) {
    if (row->chars[j] == '\t')
      rx += (KILO_TAB_STOP - 1) - (rx % KILO_TAB_STOP);
    rx++;
  }
  return rx;
}

void editorUpdateRow(erow *row) { … }

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

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

compiles, but with no observable effects

For each character, if it's a tab we use `rx % KILO_TAB_STOP` to find out how many columns we are to the right of the last tab stop, and then subtract that from `KILO_TAB_STOP - 1` to find out how many columns we are to the left of the next tab stop. We add that amount to `rx` to get just to the left of the next tab stop, and then the unconditional `rx++` statement gets us right on the next tab stop. Notice how this works even if we are currently on a tab stop.

Let's call `editorRowCxToRx()` at the top of `editorScroll()` to finally set `E.rx` to its proper value.

kilo.c — Step 89 — set-rx

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

void editorScroll() {
  E.rx = 0;
  if (E.cy < E.numrows) {
    E.rx = editorRowCxToRx(&E.row[E.cy], E.cx);
  }

  if (E.cy < E.rowoff) {
    E.rowoff = E.cy;
  }
  if (E.cy >= E.rowoff + E.screenrows) {
    E.rowoff = E.cy - E.screenrows + 1;
  }
  if (E.rx < E.coloff) {
    E.coloff = E.rx;
  }
  if (E.rx >= E.coloff + E.screencols) {
    E.coloff = E.rx - E.screencols + 1;
  }
}

void editorDrawRows(struct abuf *ab) { … }

void editorRefreshScreen() { … }

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

↗ compiles

You should now be able to confirm that the cursor moves properly within lines that contain tabs.

Scrolling with `Page Up` and `Page Down`

Now that we have scrolling, let's make the `Page Up` and `Page Down` keys scroll up or down an entire page.

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row 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:
      E.cx = E.screencols - 1;
      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;
  }
}

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

↗ compiles

To scroll up or down a page, we position the cursor either at the top or bottom of the screen, and then simulate an entire screen's worth of `↑` or `↓` keypresses. Delegating to `editorMoveCursor()` takes care of all the bounds-checking and cursor-fixing that needs to be done when moving the cursor.

Move to the end of the line with `End`

Now let's have the `End` key move the cursor to the end of the current line. (The `Home` key already moves the cursor to the beginning of the line, since we made `E.cx` relative to the file instead of relative to the screen.)

kilo.c — Step 91 — end-key

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row 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;
  }
}

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

↗ compiles

The `End` key brings the cursor to the end of the current line. If there is no current line, then `E.cx` must be `0` and it should stay at `0`, so there's nothing to do.

Status bar

The last thing we'll add before finally getting to text editing is a status bar. This will show useful information such as the filename, how many lines are in the file, and what line you're currently on. Later we'll add a marker that tells you whether the file has been modified since it was last saved, and we'll also display the filetype when we implement syntax highlighting.

First we'll simply make room for a one-line status bar at the bottom of the screen.

kilo.c **Step 92** `status-bar-make-room`

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

void editorScroll() { … }

void editorDrawRows(struct abuf *ab) {
  int y;
  for (y = 0; y < E.screenrows; y++) {
    int filerow = y + E.rowoff;
    if (filerow >= E.numrows) {
      if (E.numrows == 0 && y == E.screenrows / 3) {
        char welcome[80];
        int welcomelen = snprintf(welcome, sizeof(welcome),
          "Kilo editor -- version %s", KILO_VERSION);
        if (welcomelen > E.screencols) welcomelen = E.screencols;
        int padding = (E.screencols - welcomelen) / 2;
        if (padding) {
          abAppend(ab, "~", 1);
          padding--;
        }
        while (padding--) abAppend(ab, " ", 1);
        abAppend(ab, welcome, welcomelen);
      } else {
        abAppend(ab, "~", 1);
      }
    } else {
      int len = E.row[filerow].rsize - E.coloff;
      if (len < 0) len = 0;
      if (len > E.screencols) len = E.screencols;
      abAppend(ab, &E.row[filerow].render[E.coloff], len);
    }

    abAppend(ab, "\x1b[K", 3);
    if (y < E.screenrows - 1) {
    abAppend(ab, "\r\n", 2);
  }
  }
}
```

```
void editorRefreshScreen() { … }

/*** 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;

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

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

↗ compiles

We decrement `E.screenrows` so that `editorDrawRows()` doesn't try to draw a line of text at the bottom of the screen. We also have `editorDrawRows()` print a newline after the last row it draws, since the status bar is now the final line being drawn on the screen.

Notice how with those two changes, our text viewer works just fine, including scrolling and cursor movement, and the last line where our status bar will be is left alone by the rest of the display code.

To make the status bar stand out, we're going to display it with inverted colors: black text on a white background. The escape sequence `<esc>[7m` switches to inverted colors, and `<esc>[m` switches back to normal formatting. Let's draw a blank white status bar of inverted space characters.

kilo.c **Step 93** `blank-status-bar`

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

```
void editorScroll() { … }

void editorDrawRows(struct abuf *ab) { … }

void editorDrawStatusBar(struct abuf *ab) {
  abAppend(ab, "\x1b[7m", 4);
  int len = 0;
  while (len < E.screencols) {
    abAppend(ab, " ", 1);
    len++;
  }
  abAppend(ab, "\x1b[m", 3);
}

void editorRefreshScreen() {
  editorScroll();

  struct abuf ab = ABUF_INIT;

  abAppend(&ab, "\x1b[?25l", 6);
  abAppend(&ab, "\x1b[H", 3);

  editorDrawRows(&ab);
  editorDrawStatusBar(&ab);

  char buf[32];
  snprintf(buf, sizeof(buf), "\x1b[%d;%dH", (E.cy - E.rowoff) + 1,
                                            (E.rx - E.coloff) + 1);
  abAppend(&ab, buf, strlen(buf));

  abAppend(&ab, "\x1b[?25h", 6);

  write(STDOUT_FILENO, ab.b, ab.len);
  abFree(&ab);
}

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

↗ compiles

The `m` command (Select Graphic Rendition) causes the text printed after it to be printed with various possible attributes including bold (`1`), underscore (`4`), blink (`5`), and inverted colors (`7`). For example, you could specify all of these attributes using the command `<esc>[1;4;5;7m`. An argument of `0` clears all attributes, and is the default argument, so we use `<esc>[m` to go back to normal text formatting.

Since we want to display the filename in the status bar, let's add a `filename` string to the global editor state, and save a copy of the filename there when a file is opened.

kilo.c — Step 94 — filename

```
/*** 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;
  char *filename;
  struct termios orig_termios;
};

struct editorConfig E;

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

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);
}
```

```
/*** 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.filename = NULL;

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

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

compiles, but with no observable effects

`strdup()` comes from `<string.h>`. It makes a copy of the given string, allocating the required memory and assuming you will `free()` that memory.

We initialize `E.filename` to the `NULL` pointer, and it will stay `NULL` if a file isn't opened (which is what happens when the program is run without arguments).

Now we're ready to display some information in the status bar. We'll display up to 20 characters of the filename, followed by the number of lines in the file. If there is no filename, we'll display `[No Name]` instead.

kilo.c **Step 95** `status-bar-left`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row 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];
  int len = snprintf(status, sizeof(status), "%.20s - %d lines",
    E.filename ? E.filename : "[No Name]", E.numrows);
  if (len > E.screencols) len = E.screencols;
  abAppend(ab, status, len);
  while (len < E.screencols) {
    abAppend(ab, " ", 1);
    len++;
  }
  abAppend(ab, "\x1b[m", 3);
}

void editorRefreshScreen() { … }

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

↗ compiles

We make sure to cut the status string short in case it doesn't fit inside the width of the window. Notice how we still use the code that draws spaces up to the end of the screen, so that the entire status bar has a white background.

Now let's show the current line number, and align it to the right edge of the screen.

kilo.c — Step 96 — status-bar-right

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row 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",
    E.filename ? E.filename : "[No Name]", E.numrows);
  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);
}

void editorRefreshScreen() { … }

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

↗ compiles

The current line is stored in `E.cy`, which we add `1` to since `E.cy` is 0-indexed. After printing the first status string, we want to keep printing spaces until we get to the point where if we printed the second status string, it would end up against the right edge of the screen. That happens when `E.screencols - len` is equal to the length of the second status string. At that point we print the status string and break out of the loop, as the entire status bar has now been printed.

Status message

We're going to add one more line below our status bar. This will be for displaying messages to the user, and prompting the user for input when doing a search, for example. We'll store the current message in a string called `statusmsg`, which we'll put in the global editor state. We'll also store a timestamp for the message, so that we can erase it a few seconds after it's been displayed.

kilo.c — Step 97 — status-message

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

#define _DEFAULT_SOURCE
#define _BSD_SOURCE
#define _GNU_SOURCE

#include <ctype.h>
#include <errno.h>
#include <stdio.h>
```

```c
#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 ***/

typedef struct erow { … } erow;

struct editorConfig {
  int cx, cy;
  int rx;
  int rowoff;
  int coloff;
  int screenrows;
  int screencols;
  int numrows;
  erow *row;
  char *filename;
  char statusmsg[80];
  time_t statusmsg_time;
  struct termios orig_termios;
};

struct editorConfig E;

/*** terminal ***/
/*** row 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.filename = NULL;
  E.statusmsg[0] = '\0';
  E.statusmsg_time = 0;
```

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

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

compiles, but with no observable effects

`time_t` comes from `<time.h>`.

We initialize `E.statusmsg` to an empty string, so no message will be displayed by default. `E.statusmsg_time` will contain the timestamp when we set a status message.

Let's define an `editorSetStatusMessage()` function. This function will take a format string and a variable number of arguments, like the `printf()` family of functions.

kilo.c — Step 98 — set-status-message

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

#define _DEFAULT_SOURCE
#define _BSD_SOURCE
#define _GNU_SOURCE

#include <ctype.h>
#include <errno.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 ***/
/*** file i/o ***/
/*** append buffer ***/
/*** output ***/

void editorScroll() { … }
```

```
void editorDrawRows(struct abuf *ab) { … }

void editorDrawStatusBar(struct abuf *ab) { … }

void editorRefreshScreen() { … }

void editorSetStatusMessage(const char *fmt, ...) {
  va_list ap;
  va_start(ap, fmt);
  vsnprintf(E.statusmsg, sizeof(E.statusmsg), fmt, ap);
  va_end(ap);
  E.statusmsg_time = time(NULL);
}

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

void initEditor() { … }

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

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

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

  return 0;
}
```

compiles, but with no observable effects

`va_list`, `va_start()`, and `va_end()` come from `<stdarg.h>`. `vsnprintf()` comes from `<stdio.h>`. `time()` comes from `<time.h>`.

In `main()`, we set the initial status message to a help message with the key bindings that our text editor uses (currently, just `Ctrl-Q` to quit).

`vsnprintf()` helps us make our own `printf()`-style function. We store the resulting string in `E.statusmsg`, and set `E.statusmsg_time` to the current time,

which can be gotten by passing `NULL` to `time()`. (It returns the number of seconds that have passed since midnight, January 1, 1970 as an integer.)

The `...` argument makes `editorSetStatusMessage()` a variadic function, meaning it can take any number of arguments. C's way of dealing with these arguments is by having you call `va_start()` and `va_end()` on a value of type `va_list`. The last argument before the `...` (in this case, `fmt`) must be passed to `va_start()`, so that the address of the next arguments is known. Then, between the `va_start()` and `va_end()` calls, you would call `va_arg()` and pass it the type of the next argument (which you usually get from the given format string) and it would return the value of that argument. In this case, we pass `fmt` and `ap` to `vsnprintf()` and it takes care of reading the format string and calling `va_arg()` to get each argument.

Now that we have a status message to display, let's make room for a second line beneath our status bar where we'll display the message.

kilo.c **Step 99** `message-bar-make-room`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** row 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",
    E.filename ? E.filename : "[No Name]", E.numrows);
  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 editorRefreshScreen() { … }

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

/*** 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.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

We decrement `E.screenrows` again, and print a newline after the first status bar. We now have a blank final line once again.

Let's draw the message bar in a new `editorDrawMessageBar()` function.

kilo.c **Step 100** `draw-message-bar`

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

```
void editorScroll() { … }

void editorDrawRows(struct abuf *ab) { … }

void editorDrawStatusBar(struct abuf *ab) { … }

void editorDrawMessageBar(struct abuf *ab) {
  abAppend(ab, "\x1b[K", 3);
  int msglen = strlen(E.statusmsg);
  if (msglen > E.screencols) msglen = E.screencols;
  if (msglen && time(NULL) - E.statusmsg_time < 5)
    abAppend(ab, E.statusmsg, msglen);
}

void editorRefreshScreen() {
  editorScroll();

  struct abuf ab = ABUF_INIT;

  abAppend(&ab, "\x1b[?25l", 6);
  abAppend(&ab, "\x1b[H", 3);

  editorDrawRows(&ab);
  editorDrawStatusBar(&ab);
  editorDrawMessageBar(&ab);

  char buf[32];
  snprintf(buf, sizeof(buf), "\x1b[%d;%dH", (E.cy - E.rowoff) + 1,
                                            (E.rx - E.coloff) + 1);
  abAppend(&ab, buf, strlen(buf));

  abAppend(&ab, "\x1b[?25h", 6);

  write(STDOUT_FILENO, ab.b, ab.len);
  abFree(&ab);
}

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

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

↗ compiles

First we clear the message bar with the `<esc>[K` escape sequence. Then we make sure the message will fit the width of the screen, and then display the message, but only if the message is less than 5 seconds old.

When you start up the program now, you should see the help message at the bottom. It will disappear *when you press a key* after 5 seconds. Remember, we only refresh the screen after each keypress.

In the next chapter, we will turn our text viewer into a text editor, allowing the user to insert and delete characters and save their changes to disk.