← prev contents next →

Raw input and output

Press `Ctrl-Q` to quit

Last chapter we saw that the `Ctrl` key combined with the alphabetic keys seemed to map to bytes 1–26. We can use this to detect `Ctrl` key combinations and map them to different operations in our editor. We'll start by mapping `Ctrl-Q` to the quit operation.

kilo.c **Step 20** ctrl-q

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

#include <ctype.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#include <unistd.h>

/*** defines ***/

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

/*** data ***/
/*** terminal ***/
/*** init ***/

int main() {
  enableRawMode();

  while (1) {
    char c = '\0';
    if (read(STDIN_FILENO, &c, 1) == -1 && errno != EAGAIN) die("read");
    if (iscntrl(c)) {
      printf("%d\r\n", c);
    } else {
      printf("%d ('%c')\r\n", c, c);
    }
    if (c == CTRL_KEY('q')) break;
  }

  return 0;
}
```

↗ compiles

The `CTRL_KEY` macro bitwise-ANDs a character with the value `00011111`, in binary. (In C, you generally specify bitmasks using hexadecimal, since C doesn't have binary literals, and hexadecimal is more concise and readable once you get used to it.) In other words, it sets the upper 3 bits of the character to `0`. This mirrors what the `Ctrl` key does in the terminal: it strips bits 5 and 6 from whatever key you press in combination with `Ctrl`, and sends that. (By convention, bit numbering starts from 0.) The ASCII character set seems to be designed this way on purpose. (It is also similarly designed so that you can set and clear bit 5 to switch between lowercase and uppercase.)

Refactor keyboard input

Let's make a function for low-level keypress reading, and another function for mapping keypresses to editor operations. We'll also stop printing out keypresses at this point.

kilo.c **Step 21** `refactor-input`

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

void die(const char *s) { … }

void disableRawMode() { … }

void enableRawMode() { … }

char editorReadKey() {
  int nread;
  char c;
  while ((nread = read(STDIN_FILENO, &c, 1)) != 1) {
    if (nread == -1 && errno != EAGAIN) die("read");
  }
  return c;
}

/*** input ***/

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

  switch (c) {
    case CTRL_KEY('q'):
      exit(0);
      break;
  }
```

```
}

/*** init ***/

int main() {
  enableRawMode();

  while (1) {
    editorProcessKeypress();
  }

  return 0;
}
```

↗ compiles

`editorReadKey()`'s job is to wait for one keypress, and return it. Later, we'll expand this function to handle escape sequences, which involves reading multiple bytes that represent a single keypress, as is the case with the arrow keys.

`editorProcessKeypress()` waits for a keypress, and then handles it. Later, it will map various `Ctrl` key combinations and other special keys to different editor functions, and insert any alphanumeric and other printable keys' characters into the text that is being edited.

Note that `editorReadKey()` belongs in the `/*** terminal ***/` section because it deals with low-level terminal input, whereas `editorProcessKeypress()` belongs in the new `/*** input ***/` section because it deals with mapping keys to editor functions at a much higher level.

Now we have vastly simplified `main()`, and we will try to keep it that way.

Clear the screen

We're going to render the editor's user interface to the screen after each keypress. Let's start by just clearing the screen.

kilo.c **Step 22** `clear-screen`

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

void die(const char *s) { … }
```

```
void disableRawMode() { … }

void enableRawMode() { … }

char editorReadKey() { … }

/*** output ***/

void editorRefreshScreen() {
  write(STDOUT_FILENO, "\x1b[2J", 4);
}

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

int main() {
  enableRawMode();

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

  return 0;
}
```

↗ compiles

`write()` and `STDOUT_FILENO` come from `<unistd.h>`.

The `4` in our `write()` call means we are writing `4` bytes out to the terminal. The first byte is `\x1b`, which is the escape character, or `27` in decimal. (Try and remember `\x1b`, we will be using it a lot.) The other three bytes are `[2J`.

We are writing an *escape sequence* to the terminal. Escape sequences always start with an escape character (`27`) followed by a `[` character. Escape sequences instruct the terminal to do various text formatting tasks, such as coloring text, moving the cursor around, and clearing parts of the screen.

We are using the `J` command (Erase In Display) to clear the screen. Escape sequence commands take arguments, which come before the command. In this case the argument is `2`, which says to clear the entire screen. `<esc>[1J` would clear the screen up to where the cursor is, and `<esc>[0J` would clear the screen from the cursor up to the end of the screen. Also, `0` is the default argument for `J`, so just `<esc>[J` by itself would also clear the screen from the cursor to the end.

For our text editor, we will be mostly using VT100 escape sequences, which are supported very widely by modern terminal emulators. See the VT100 User Guide for complete documentation of each escape sequence.

If we wanted to support the maximum number of terminals out there, we could use the ncurses library, which uses the terminfo database to figure out the capabilities of a terminal and what escape sequences to use for that particular terminal.

Reposition the cursor

You may notice that the `<esc>[2J` command left the cursor at the bottom of the screen. Let's reposition it at the top-left corner so that we're ready to draw the editor interface from top to bottom.

kilo.c **Step 23** `cursor-home`

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

void editorRefreshScreen() {
  write(STDOUT_FILENO, "\x1b[2J", 4);
  write(STDOUT_FILENO, "\x1b[H", 3);
}

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

↗ compiles

This escape sequence is only `3` bytes long, and uses the `H` command (Cursor Position) to position the cursor. The `H` command actually takes two arguments: the row number and the column number at which to position the cursor. So if you have an 80×24 size terminal and you want the cursor in the center of the screen, you could use the command `<esc>[12;40H`. (Multiple arguments are separated by a `;` character.) The default arguments for `H` both happen to be `1`, so we can leave both arguments out and it will position the cursor at the first row and first column, as if we had sent the `<esc>[1;1H` command. (Rows and columns are numbered starting at `1`, not `0`.)

Clear the screen on exit

Let's clear the screen and reposition the cursor when our program exits. If an error

occurs in the middle of rendering the screen, we don't want a bunch of garbage left over on the screen, and we don't want the error to be printed wherever the cursor happens to be at that point.

kilo.c **Step 24** `clean-exit`

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

void die(const char *s) {
  write(STDOUT_FILENO, "\x1b[2J", 4);
  write(STDOUT_FILENO, "\x1b[H", 3);

  perror(s);
  exit(1);
}

void disableRawMode() { … }

void enableRawMode() { … }

char editorReadKey() { … }

/*** output ***/
/*** input ***/

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

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

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

↗ compiles

We have two exit points we want to clear the screen at: `die()`, and when the user presses `Ctrl-Q` to quit.

We could use `atexit()` to clear the screen when our program exits, but then the error

message printed by `die()` would get erased right after printing it.

Tildes

It's time to start drawing. Let's draw a column of tildes (`~`) on the left hand side of the screen, like vim does. In our text editor, we'll draw a tilde at the beginning of any lines that come after the end of the file being edited.

kilo.c **Step 25** `tildes`

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

void editorDrawRows() {
  int y;
  for (y = 0; y < 24; y++) {
    write(STDOUT_FILENO, "~\r\n", 3);
  }
}

void editorRefreshScreen() {
  write(STDOUT_FILENO, "\x1b[2J", 4);
  write(STDOUT_FILENO, "\x1b[H", 3);

  editorDrawRows();

  write(STDOUT_FILENO, "\x1b[H", 3);
}

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

↗ compiles

`editorDrawRows()` will handle drawing each row of the buffer of text being edited. For now it draws a tilde in each row, which means that row is not part of the file and can't contain any text.

We don't know the size of the terminal yet, so we don't know how many rows to draw. For now we just draw `24` rows.

After we're done drawing, we do another `<esc>[H` escape sequence to reposition the cursor back up at the top-left corner.

Global state

Our next goal is to get the size of the terminal, so we know how many rows to draw in `editorDrawRows()`. But first, let's set up a global struct that will contain our editor state, which we'll use to store the width and height of the terminal. For now, let's just put our `orig_termios` global into the struct.

kilo.c **Step 26** `global-state`

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

struct editorConfig {
  struct termios orig_termios;
};

struct editorConfig E;

/*** terminal ***/

void die(const char *s) { … }

void disableRawMode() {
  if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &E.orig_termios) == -1)
    die("tcsetattr");
}

void enableRawMode() {
  if (tcgetattr(STDIN_FILENO, &E.orig_termios) == -1) die("tcgetattr");
  atexit(disableRawMode);

  struct termios raw = E.orig_termios;
  raw.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);
  raw.c_oflag &= ~(OPOST);
  raw.c_cflag |= (CS8);
  raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);
  raw.c_cc[VMIN] = 0;
  raw.c_cc[VTIME] = 1;

  if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw) == -1) die("tcsetattr");
}

char editorReadKey() { … }

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

compiles, but with no observable effects

Our global variable containing our editor state is named `E`. We must replace all occurrences of `orig_termios` with `E.orig_termios`.

Window size, the easy way

On most systems, you should be able to get the size of the terminal by simply calling `ioctl()` with the `TIOCGWINSZ` request. (As far as I can tell, it stands for **T**erminal **IOC**tl (which itself stands for **I**nput/**O**utput **C**on**t**ro**l**) **G**et **WIN**dow **S**i**Z**e.)

kilo.c **Step 27** `ioctl`

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

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

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

void die(const char *s) { … }

void disableRawMode() { … }

void enableRawMode() { … }

char editorReadKey() { … }

int getWindowSize(int *rows, int *cols) {
  struct winsize ws;

  if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1 || ws.ws_col == 0) {
    return -1;
  } else {
    *cols = ws.ws_col;
    *rows = ws.ws_row;
    return 0;
  }
}
```

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

compiles, but with no observable effects

`ioctl()`, `TIOCGWINSZ`, and `struct winsize` come from `<sys/ioctl.h>`.

On success, `ioctl()` will place the number of columns wide and the number of rows high the terminal is into the given `winsize` struct. On failure, `ioctl()` returns `-1`. We also check to make sure the values it gave back weren't `0`, because apparently that's a possible erroneous outcome. If `ioctl()` failed in either way, we have `getWindowSize()` report failure by returning `-1`. If it succeeded, we pass the values back by setting the `int` references that were passed to the function. (This is a common approach to having functions return multiple values in C. It also allows you to use the return value to indicate success or failure.)

Now let's add `screenrows` and `screencols` to our global editor state, and call `getWindowSize()` to fill in those values.

kilo.c **Step 28** `init-editor`

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

struct editorConfig {
  int screenrows;
  int screencols;
  struct termios orig_termios;
};

struct editorConfig E;

/*** terminal ***/
/*** output ***/
/*** input ***/
/*** init ***/

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

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

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

  return 0;
}
```

compiles, but with no observable effects

`initEditor()`'s job will be to initialize all the fields in the `E` struct.

Now we're ready to display the proper number of tildes on the screen:

kilo.c **Step 29** `screenrows`

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

void editorDrawRows() {
  int y;
  for (y = 0; y < E.screenrows; y++) {
    write(STDOUT_FILENO, "~\r\n", 3);
  }
}

void editorRefreshScreen() { … }

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

compiles

Window size, the hard way

`ioctl()` isn't guaranteed to be able to request the window size on all systems, so we are going to provide a fallback method of getting the window size.

The strategy is to position the cursor at the bottom-right of the screen, then use escape sequences that let us query the position of the cursor. That tells us how many rows and columns there must be on the screen.

Let's start by moving the cursor to the bottom-right.

kilo.c **Step 30** `bottom-right`

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

void die(const char *s) { … }

void disableRawMode() { … }

void enableRawMode() { … }

char editorReadKey() { … }

int getWindowSize(int *rows, int *cols) {
  struct winsize ws;

  if (1 || ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1 || ws.ws_col == 0) {
    if (write(STDOUT_FILENO, "\x1b[999C\x1b[999B", 12) != 12) return -1;
    editorReadKey();
    return -1;
  } else {
    *cols = ws.ws_col;
    *rows = ws.ws_row;
    return 0;
  }
}

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

↗ compiles

As you might have gathered from the code, there is no simple “move the cursor to the bottom-right corner” command.

We are sending two escape sequences one after the other. The `C` command (Cursor Forward) moves the cursor to the right, and the `B` command (Cursor Down) moves the cursor down. The argument says how much to move it right or down by. We use a very large value, `999`, which should ensure that the cursor reaches the right and bottom edges of the screen.

The `C` and `B` commands are specifically documented to stop the cursor from going past the edge of the screen. The reason we don’t use the `<esc>[999;999H` command is that the documentation doesn’t specify what happens when you try to move the cursor off-screen.

Note that we are sticking a `1 ||` at the front of our `if` condition temporarily, so that we can test this fallback branch we are developing.

Because we're always returning `-1` (meaning an error occurred) from `getWindowSize()` at this point, we make a call to `editorReadKey()` so we can observe the results of our escape sequences before the program calls `die()` and clears the screen. When you run the program, you should see the cursor is positioned at the bottom-right corner of the screen, and then when you press a key you'll see the error message printed by `die()` after it clears the screen.

Next we need to get the cursor position. The `n` command (Device Status Report) can be used to query the terminal for status information. We want to give it an argument of `6` to ask for the cursor position. Then we can read the reply from the standard input. Let's print out each character from the standard input to see what the reply looks like.

kilo.c **Step 31** cursor-query

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

void die(const char *s) { … }

void disableRawMode() { … }

void enableRawMode() { … }

char editorReadKey() { … }

int getCursorPosition(int *rows, int *cols) {
  if (write(STDOUT_FILENO, "\x1b[6n", 4) != 4) return -1;

  printf("\r\n");
  char c;
  while (read(STDIN_FILENO, &c, 1) == 1) {
    if (iscntrl(c)) {
      printf("%d\r\n", c);
    } else {
      printf("%d ('%c')\r\n", c, c);
    }
  }

  editorReadKey();

  return -1;
```

```
}

int getWindowSize(int *rows, int *cols) {
  struct winsize ws;

  if (1 || ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1 || ws.ws_col == 0) {
    if (write(STDOUT_FILENO, "\x1b[999C\x1b[999B", 12) != 12) return -1;
    return getCursorPosition(rows, cols);
  } else {
    *cols = ws.ws_col;
    *rows = ws.ws_row;
    return 0;
  }
}
```

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

↗ compiles

The reply is an escape sequence! It's an escape character (`27`), followed by a `[` character, and then the actual response: `24;80R`, or similar. (This escape sequence is documented as Cursor Position Report.)

As before, we've inserted a temporary call to `editorReadKey()` to let us observe our debug output before the screen gets cleared on exit.

(Note: If you're using **Bash on Windows**, `read()` doesn't time out so you'll be stuck in an infinite loop. You'll have to kill the process externally, or exit and reopen the command prompt window.)

We're going to have to parse this response. But first, let's read it into a buffer. We'll keep reading characters until we get to the `R` character.

kilo.c **Step 32** `response-buffer`

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

void die(const char *s) { … }

void disableRawMode() { … }

void enableRawMode() { … }
```

```
char editorReadKey() { … }

int getCursorPosition(int *rows, int *cols) {
  char buf[32];
  unsigned int i = 0;

  if (write(STDOUT_FILENO, "\x1b[6n", 4) != 4) return -1;

  while (i < sizeof(buf) - 1) {
    if (read(STDIN_FILENO, &buf[i], 1) != 1) break;
    if (buf[i] == 'R') break;
    i++;
  }
  buf[i] = '\0';

  printf("\r\n&buf[1]: '%s'\r\n", &buf[1]);

  editorReadKey();

  return -1;
}

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

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

↗ compiles

When we print out the buffer, we don't want to print the `'\x1b'` character, because the terminal would interpret it as an escape sequence and wouldn't display it. So we skip the first character in `buf` by passing `&buf[1]` to `printf()`. `printf()` expects strings to end with a `0` byte, so we make sure to assign `'\0'` to the final byte of `buf`.

If you run the program, you'll see we have the response in `buf` in the form of `<esc>[24;80`. Let's parse the two numbers out of there using `sscanf()`:

kilo.c **Step 33** `parse-response`

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

void die(const char *s) { … }
```

```
void disableRawMode() { … }

void enableRawMode() { … }

char editorReadKey() { … }

int getCursorPosition(int *rows, int *cols) {
  char buf[32];
  unsigned int i = 0;

  if (write(STDOUT_FILENO, "\x1b[6n", 4) != 4) return -1;

  while (i < sizeof(buf) - 1) {
    if (read(STDIN_FILENO, &buf[i], 1) != 1) break;
    if (buf[i] == 'R') break;
    i++;
  }
  buf[i] = '\0';

  if (buf[0] != '\x1b' || buf[1] != '[') return -1;
  if (sscanf(&buf[2], "%d;%d", rows, cols) != 2) return -1;

  return 0;
}

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

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

↗ compiles

`sscanf()` comes from `<stdio.h>`.

First we make sure it responded with an escape sequence. Then we pass a pointer to the third character of `buf` to `sscanf()`, skipping the `'\x1b'` and `'['` characters. So we are passing a string of the form `24;80` to `sscanf()`. We are also passing it the string `%d;%d` which tells it to parse two integers separated by a `;`, and put the values into the `rows` and `cols` variables.

Our fallback method for getting the window size is now complete. You should see that `editorDrawRows()` prints the correct number of tildes for the height of your terminal.

Now that we know that works, let's remove the `1 ||` we put in the `if` condition temporarily.

kilo.c **Step 34** back-to-ioctl

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

void die(const char *s) { … }

void disableRawMode() { … }

void enableRawMode() { … }

char editorReadKey() { … }

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

int getWindowSize(int *rows, int *cols) {
  struct winsize ws;

  if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == -1 || ws.ws_col == 0) {
    if (write(STDOUT_FILENO, "\x1b[999C\x1b[999B", 12) != 12) return -1;
    return getCursorPosition(rows, cols);
  } else {
    *cols = ws.ws_col;
    *rows = ws.ws_row;
    return 0;
  }
}

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

compiles, but with no observable effects

The last line

Maybe you noticed the last line of the screen doesn't seem to have a tilde. That's because of a small bug in our code. When we print the final tilde, we then print a `"\r\n"` like on any other line, but this causes the terminal to scroll in order to make room for a new, blank line. Let's make the last line an exception when we print our `"\r\n"`'s.

kilo.c **Step 35** last-line

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

```
/*** data ***/
/*** terminal ***/
/*** output ***/

void editorDrawRows() {
  int y;
  for (y = 0; y < E.screenrows; y++) {
    write(STDOUT_FILENO, "~", 1);

    if (y < E.screenrows - 1) {
      write(STDOUT_FILENO, "\r\n", 2);
    }
  }
}

void editorRefreshScreen() { … }

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

↗ compiles

Append buffer

It's not a good idea to make a whole bunch of small `write()`'s every time we refresh the screen. It would be better to do one big `write()`, to make sure the whole screen updates at once. Otherwise there could be small unpredictable pauses between `write()`'s, which would cause an annoying flicker effect.

We want to replace all our `write()` calls with code that appends the string to a buffer, and then `write()` this buffer out at the end. Unfortunately, C doesn't have dynamic strings, so we'll create our own dynamic string type that supports one operation: appending.

Let's start by making a new `/*** append buffer ***/` section, and defining the `abuf` struct under it.

kilo.c **Step 36** `abuf-struct`

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

void die(const char *s) { … }

void disableRawMode() { … }
```

```
void enableRawMode() { … }

char editorReadKey() { … }

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

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

/*** append buffer ***/

struct abuf {
  char *b;
  int len;
};

#define ABUF_INIT {NULL, 0}

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

compiles, but with no observable effects

An append buffer consists of a pointer to our buffer in memory, and a length. We define an `ABUF_INIT` constant which represents an empty buffer. This acts as a constructor for our `abuf` type.

Next, let's define the `abAppend()` operation, as well as the `abFree()` destructor.

kilo.c **Step 37** `abuf-append`

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

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

/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** append buffer ***/

struct abuf { … };
```

```
#define ABUF_INIT {NULL, 0}

void abAppend(struct abuf *ab, const char *s, int len) {
  char *new = realloc(ab->b, ab->len + len);

  if (new == NULL) return;
  memcpy(&new[ab->len], s, len);
  ab->b = new;
  ab->len += len;
}

void abFree(struct abuf *ab) {
  free(ab->b);
}
```

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

compiles, but with no observable effects

`realloc()` and `free()` come from `<stdlib.h>`. `memcpy()` comes from `<string.h>`.

To append a string `s` to an `abuf`, the first thing we do is make sure we allocate enough memory to hold the new string. We ask `realloc()` to give us a block of memory that is the size of the current string plus the size of the string we are appending. `realloc()` will either extend the size of the block of memory we already have allocated, or it will take care of `free()`ing the current block of memory and allocating a new block of memory somewhere else that is big enough for our new string.

Then we use `memcpy()` to copy the string `s` after the end of the current data in the buffer, and we update the pointer and length of the `abuf` to the new values.

`abFree()` is a destructor that deallocates the dynamic memory used by an `abuf`.

Okay, our `abuf` type is ready to be put to use.

kilo.c **Step 38** `use-abuf`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** append buffer ***/
/*** output ***/
```

```
void editorDrawRows(struct abuf *ab) {
  int y;
  for (y = 0; y < E.screenrows; y++) {
    abAppend(ab, "~", 1);

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

void editorRefreshScreen() {
  struct abuf ab = ABUF_INIT;

  abAppend(&ab, "\x1b[2J", 4);
  abAppend(&ab, "\x1b[H", 3);

  editorDrawRows(&ab);

  abAppend(&ab, "\x1b[H", 3);

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

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

compiles, but with no observable effects

In `editorRefreshScreen()`, we first initialize a new `abuf` called `ab`, by assigning `ABUF_INIT` to it. We then replace each occurrence of `write(STDOUT_FILENO, ...)` with `abAppend(&ab, ...)`. We also pass `ab` into `editorDrawRows()`, so it too can use `abAppend()`. Lastly, we `write()` the buffer's contents out to standard output, and free the memory used by the `abuf`.

Hide the cursor when repainting

There is another possible source of the annoying flicker effect we will take care of now. It's possible that the cursor might be displayed in the middle of the screen somewhere for a split second while the terminal is drawing to the screen. To make sure that doesn't happen, let's hide the cursor before refreshing the screen, and show it again immediately after the refresh finishes.

kilo.c **Step 39** `hide-cursor`

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

```
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** append buffer ***/
/*** output ***/

void editorDrawRows(struct abuf *ab) { … }

void editorRefreshScreen() {
  struct abuf ab = ABUF_INIT;

  abAppend(&ab, "\x1b[?25l", 6);
  abAppend(&ab, "\x1b[2J", 4);
  abAppend(&ab, "\x1b[H", 3);

  editorDrawRows(&ab);

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

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

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

compiles, but with no observable effects

We use escape sequences to tell the terminal to hide and show the cursor. The `h` and `l` commands (Set Mode, Reset Mode) are used to turn on and turn off various terminal features or "modes". The VT100 User Guide just linked to doesn't document argument `?25` which we use above. It appears the cursor hiding/showing feature appeared in later VT models. So some terminals might not support hiding/showing the cursor, but if they don't, then they will just ignore those escape sequences, which isn't a big deal in this case.

Clear lines one at a time

Instead of clearing the entire screen before each refresh, it seems more optimal to clear each line as we redraw them. Let's remove the `<esc>[2J` (clear entire screen) escape sequence, and instead put a `<esc>[K` sequence at the end of each line we draw.

kilo.c **Step 40** `clear-line`

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

```
/*** data ***/
/*** terminal ***/
/*** append buffer ***/
/*** output ***/

void editorDrawRows(struct abuf *ab) {
  int y;
  for (y = 0; y < E.screenrows; y++) {
    abAppend(ab, "~", 1);

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

void editorRefreshScreen() {
  struct abuf ab = ABUF_INIT;

  abAppend(&ab, "\x1b[?25l", 6);
  abAppend(&ab, "\x1b[2J", 4);
  abAppend(&ab, "\x1b[H", 3);

  editorDrawRows(&ab);

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

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

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

compiles, but with no observable effects

The `K` command (Erase In Line) erases part of the current line. Its argument is analogous to the `J` command's argument: `2` erases the whole line, `1` erases the part of the line to the left of the cursor, and `0` erases the part of the line to the right of the cursor. `0` is the default argument, and that's what we want, so we leave out the argument and just use `<esc>[K`.

Welcome message

Perhaps it's time to display a welcome message. Let's display the name of our editor

and a version number a third of the way down the screen.

kilo.c **Step 41** welcome

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

#define KILO_VERSION "0.0.1"

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

/*** data ***/
/*** terminal ***/
/*** append buffer ***/
/*** output ***/

void editorDrawRows(struct abuf *ab) {
  int y;
  for (y = 0; y < E.screenrows; y++) {
    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;
      abAppend(ab, welcome, welcomelen);
    } else {
      abAppend(ab, "~", 1);
    }

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

void editorRefreshScreen() { … }

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

↗ compiles

`snprintf()` comes from `<stdio.h>`.

We use the `welcome` buffer and `snprintf()` to interpolate our `KILO_VERSION` string into the welcome message. We also truncate the length of the string in case the terminal is too tiny to fit our welcome message.

Now let's center it.

kilo.c **Step 42** `center`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** append buffer ***/
/*** output ***/

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

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

void editorRefreshScreen() { … }

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

↗ compiles

To center a string, you divide the screen width by `2`, and then subtract half of the string's length from that. In other words: `E.screencols/2 - welcomelen/2`, which simplifies to `(E.screencols - welcomelen) / 2`. That tells you how far from the left edge of the screen you should start printing the string. So we fill that space with space characters, except for the first character, which should be a tilde.

Move the cursor

Let's focus on input now. We want the user to be able to move the cursor around. The first step is to keep track of the cursor's `x` and `y` position in the global editor state.

kilo.c **Step 43** `cx-cy`

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

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

struct editorConfig E;

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

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

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

int main() { … }
```

compiles, but with no observable effects

`E.cx` is the horizontal coordinate of the cursor (the column) and `E.cy` is the vertical coordinate (the row). We initialize both of them to `0`, as we want the cursor to start at the top-left of the screen. (Since the C language uses indexes that start from `0`, we will use 0-indexed values wherever possible.)

Now let's add code to `editorRefreshScreen()` to move the cursor to the position stored in `E.cx` and `E.cy`.

kilo.c **Step 44** `set-cursor-position`

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

```
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** append buffer ***/
/*** output ***/

void editorDrawRows(struct abuf *ab) { … }

void editorRefreshScreen() {
  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

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

We changed the old `H` command into an `H` command with arguments, specifying the exact position we want the cursor to move to. (Make sure you deleted the old `H` command, as the above diff makes that easy to miss.)

We add `1` to `E.cy` and `E.cx` to convert from 0-indexed values to the 1-indexed values that the terminal uses.

At this point, you could try initializing `E.cx` to `10` or something, or insert `E.cx++` into the main loop, to confirm that the code works as intended so far.

Next, we'll allow the user to move the cursor using the `w` `a` `s` `d` keys. (If you're unfamiliar with using these keys as arrow keys: `w` is your up arrow, `s` is your down arrow, `a` is left, `d` is right.)

kilo.c **Step 45** `move-cursor`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** append buffer ***/
/*** output ***/
/*** input ***/

void editorMoveCursor(char key) {
  switch (key) {
    case 'a':
      E.cx--;
      break;
    case 'd':
      E.cx++;
      break;
    case 'w':
      E.cy--;
      break;
    case 's':
      E.cy++;
      break;
  }
}

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

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

    case 'w':
    case 's':
    case 'a':
    case 'd':
      editorMoveCursor(c);
      break;
  }
}

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

↗ compiles

Now you should be able to move the cursor around with those keys.

Arrow keys

Now that we have a way of mapping keypresses to move the cursor, let's replace the `w` `a` `s` `d` keys with the arrow keys. Last chapter we saw that pressing an arrow key sends multiple bytes as input to our program. These bytes are in the form of an escape sequence that starts with `'\x1b'`, `'['`, followed by an `'A'`, `'B'`, `'C'`, or `'D'` depending on which of the four arrow keys was pressed. Let's modify `editorReadKey()` to read escape sequences of this form as a single keypress.

kilo.c **Step 46** `detect-arrow-keys`

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

void die(const char *s) { … }

void disableRawMode() { … }

void enableRawMode() { … }

char editorReadKey() {
  int nread;
  char c;
  while ((nread = read(STDIN_FILENO, &c, 1)) != 1) {
    if (nread == -1 && errno != EAGAIN) die("read");
  }

  if (c == '\x1b') {
    char seq[3];

    if (read(STDIN_FILENO, &seq[0], 1) != 1) return '\x1b';
    if (read(STDIN_FILENO, &seq[1], 1) != 1) return '\x1b';

    if (seq[0] == '[') {
      switch (seq[1]) {
        case 'A': return 'w';
        case 'B': return 's';
        case 'C': return 'd';
        case 'D': return 'a';
      }
    }
```

```
    return '\x1b';
  } else {
    return c;
  }
}

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

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

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

↗ compiles

If we read an escape character, we immediately read two more bytes into the `seq` buffer. If either of these reads time out (after 0.1 seconds), then we assume the user just pressed the `Escape` key and return that. Otherwise we look to see if the escape sequence is an arrow key escape sequence. If it is, we just return the corresponding `w` `a` `s` `d` character, for now. If it's not an escape sequence we recognize, we just return the escape character.

We make the `seq` buffer 3 bytes long because we will be handling longer escape sequences in the future.

We have basically aliased the arrow keys to the `w` `a` `s` `d` keys. This gets the arrow keys working immediately, but leaves the `w` `a` `s` `d` keys still mapped to the `editorMoveCursor()` function. What we want is for `editorReadKey()` to return special values for each arrow key that let us identify that a particular arrow key was pressed.

Let's start by replacing each instance of the `w` `a` `s` `d` characters with the constants `ARROW_UP`, `ARROW_LEFT`, `ARROW_DOWN`, and `ARROW_RIGHT`.

kilo.c **Step 47** `arrow-keys-enum`

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

#define KILO_VERSION "0.0.1"

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

enum editorKey {
```

```
  ARROW_LEFT = 'a',
  ARROW_RIGHT = 'd',
  ARROW_UP = 'w',
  ARROW_DOWN = 's'
};

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

void die(const char *s) { … }

void disableRawMode() { … }

void enableRawMode() { … }

char editorReadKey() {
  int nread;
  char c;
  while ((nread = read(STDIN_FILENO, &c, 1)) != 1) {
    if (nread == -1 && errno != EAGAIN) die("read");
  }

  if (c == '\x1b') {
    char seq[3];

    if (read(STDIN_FILENO, &seq[0], 1) != 1) return '\x1b';
    if (read(STDIN_FILENO, &seq[1], 1) != 1) return '\x1b';

    if (seq[0] == '[') {
      switch (seq[1]) {
        case 'A': return ARROW_UP;
        case 'B': return ARROW_DOWN;
        case 'C': return ARROW_RIGHT;
        case 'D': return ARROW_LEFT;
      }
    }

    return '\x1b';
  } else {
    return c;
  }
}

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

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

/*** append buffer ***/
/*** output ***/
```

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

void editorMoveCursor(char key) {
  switch (key) {
    case ARROW_LEFT:
      E.cx--;
      break;
    case ARROW_RIGHT:
      E.cx++;
      break;
    case ARROW_UP:
      E.cy--;
      break;
    case ARROW_DOWN:
      E.cy++;
      break;
  }
}

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

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

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

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

compiles, but with no observable effects

Now we just have to choose a representation for arrow keys that doesn't conflict with `w` `a` `s` `d`, in the `editorKey` enum. We will give them a large integer value that is out of the range of a `char`, so that they don't conflict with any ordinary keypresses. We will also have to change all variables that store keypresses to be of type `int` instead of `char`.

kilo.c **Step 48** arrow-keys-int

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

#define KILO_VERSION "0.0.1"

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

enum editorKey {
  ARROW_LEFT = 1000,
  ARROW_RIGHT,
  ARROW_UP,
  ARROW_DOWN
};

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

void die(const char *s) { … }

void disableRawMode() { … }

void enableRawMode() { … }

int editorReadKey() {
  int nread;
  char c;
  while ((nread = read(STDIN_FILENO, &c, 1)) != 1) {
    if (nread == -1 && errno != EAGAIN) die("read");
  }

  if (c == '\x1b') {
    char seq[3];

    if (read(STDIN_FILENO, &seq[0], 1) != 1) return '\x1b';
    if (read(STDIN_FILENO, &seq[1], 1) != 1) return '\x1b';

    if (seq[0] == '[') {
      switch (seq[1]) {
        case 'A': return ARROW_UP;
        case 'B': return ARROW_DOWN;
        case 'C': return ARROW_RIGHT;
        case 'D': return ARROW_LEFT;
      }
    }

    return '\x1b';
  } else {
```

```
    return c;
  }
}

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

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

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

void editorMoveCursor(int key) {
  switch (key) {
    case ARROW_LEFT:
      E.cx--;
      break;
    case ARROW_RIGHT:
      E.cx++;
      break;
    case ARROW_UP:
      E.cy--;
      break;
    case ARROW_DOWN:
      E.cy++;
      break;
  }
}

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

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

↗ compiles

By setting the first constant in the enum to `1000`, the rest of the constants get incrementing values of `1001`, `1002`, `1003`, and so on.

That concludes our arrow key handling code. At this point, it can be fun to try entering an escape sequence manually while the program runs. Try pressing the `Escape` key, the `[` key, and `Shift+C` in sequence really fast, and you may see your keypresses being interpreted as the right arrow key being pressed. You have to be pretty fast to do it, so you may want to adjust the `VTIME` value in `enableRawMode()` temporarily, to make it easier. (It also helps to know that pressing `Ctrl-[` is the same as pressing the `Escape` key, for the same reason that `Ctrl-M` is the same as pressing `Enter`: `Ctrl` clears the 6th and 7th bits of the character you type in combination with it.)

Prevent moving the cursor off screen

Currently, you can cause the `E.cx` and `E.cy` values to go into the negatives, or go past the right and bottom edges of the screen. Let's prevent that by doing some bounds checking in `editorMoveCursor()`.

kilo.c **Step 49** `off-screen`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** 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.screenrows - 1) {
        E.cy++;
      }
      break;
  }
}

void editorProcessKeypress() { … }

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

↗ compiles

The `Page Up` and `Page Down` keys

To complete our low-level terminal code, we need to detect a few more special keypresses that use escape sequences, like the arrow keys did. We'll start with the `Page Up` and `Page Down` keys. `Page Up` is sent as `<esc>[5~` and `Page Down` is sent as `<esc>[6~`.

kilo.c **Step 50** `detect-page-up-down`

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

#define KILO_VERSION "0.0.1"

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

enum editorKey {
  ARROW_LEFT = 1000,
  ARROW_RIGHT,
  ARROW_UP,
  ARROW_DOWN,
  PAGE_UP,
  PAGE_DOWN
};

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

void die(const char *s) { … }

void disableRawMode() { … }
```

```
void enableRawMode() { … }

int editorReadKey() {
  int nread;
  char c;
  while ((nread = read(STDIN_FILENO, &c, 1)) != 1) {
    if (nread == -1 && errno != EAGAIN) die("read");
  }

  if (c == '\x1b') {
    char seq[3];

    if (read(STDIN_FILENO, &seq[0], 1) != 1) return '\x1b';
    if (read(STDIN_FILENO, &seq[1], 1) != 1) return '\x1b';

    if (seq[0] == '[') {
      if (seq[1] >= '0' && seq[1] <= '9') {
        if (read(STDIN_FILENO, &seq[2], 1) != 1) return '\x1b';
        if (seq[2] == '~') {
          switch (seq[1]) {
            case '5': return PAGE_UP;
            case '6': return PAGE_DOWN;
          }
        }
      } else {
        switch (seq[1]) {
          case 'A': return ARROW_UP;
          case 'B': return ARROW_DOWN;
          case 'C': return ARROW_RIGHT;
          case 'D': return ARROW_LEFT;
        }
      }
    }

    return '\x1b';
  } else {
    return c;
  }
}

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

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

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

compiles, but with no observable effects

Now you see why we declared `seq` to be able to store 3 bytes. If the byte after `[` is a digit, we read another byte expecting it to be a `~`. Then we test the digit byte to see if it's a `5` or a `6`.

Let's make `Page Up` and `Page Down` do something. For now, we'll have them move the cursor to the top of the screen or the bottom of the screen.

kilo.c **Step 51** `page-up-down-simple`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** 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 PAGE_UP:
    case PAGE_DOWN:
      {
        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

We create a code block with that pair of braces so that we're allowed to declare the `times` variable. (You can't declare variables directly inside a `switch` statement.) We simulate the user pressing the `↑` or `↓` keys enough times to move to the top or bottom of the screen. Implementing `Page Up` and `Page Down` in this way will make it a lot easier for us later, when we implement scrolling.

If you're on a laptop with an `Fn` key, you may be able to press `Fn`+`↑` and `Fn`+`↓` to simulate pressing the `Page Up` and `Page Down` keys.

The `Home` and `End` keys

Now let's implement the `Home` and `End` keys. Like the previous keys, these keys also send escape sequences. Unlike the previous keys, there are many different escape sequences that could be sent by these keys, depending on your OS, or your terminal emulator. The `Home` key could be sent as `<esc>[1~`, `<esc>[7~`, `<esc>[H`, or `<esc>OH`. Similarly, the `End` key could be sent as `<esc>[4~`, `<esc>[8~`, `<esc>[F`, or `<esc>OF`. Let's handle all of these cases.

kilo.c **Step 52** `detect-home-end`

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

#define KILO_VERSION "0.0.1"

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

enum editorKey {
  ARROW_LEFT = 1000,
  ARROW_RIGHT,
  ARROW_UP,
  ARROW_DOWN,
  HOME_KEY,
  END_KEY,
  PAGE_UP,
  PAGE_DOWN
};

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

void die(const char *s) { … }

void disableRawMode() { … }

void enableRawMode() { … }

int editorReadKey() {
  int nread;
  char c;
  while ((nread = read(STDIN_FILENO, &c, 1)) != 1) {
    if (nread == -1 && errno != EAGAIN) die("read");
```

```
  }

  if (c == '\x1b') {
    char seq[3];

    if (read(STDIN_FILENO, &seq[0], 1) != 1) return '\x1b';
    if (read(STDIN_FILENO, &seq[1], 1) != 1) return '\x1b';

    if (seq[0] == '[') {
      if (seq[1] >= '0' && seq[1] <= '9') {
        if (read(STDIN_FILENO, &seq[2], 1) != 1) return '\x1b';
        if (seq[2] == '~') {
          switch (seq[1]) {
            case '1': return HOME_KEY;
            case '4': return END_KEY;
            case '5': return PAGE_UP;
            case '6': return PAGE_DOWN;
            case '7': return HOME_KEY;
            case '8': return END_KEY;
          }
        }
      } else {
        switch (seq[1]) {
          case 'A': return ARROW_UP;
          case 'B': return ARROW_DOWN;
          case 'C': return ARROW_RIGHT;
          case 'D': return ARROW_LEFT;
          case 'H': return HOME_KEY;
          case 'F': return END_KEY;
        }
      }
    } else if (seq[0] == 'O') {
      switch (seq[1]) {
        case 'H': return HOME_KEY;
        case 'F': return END_KEY;
      }
    }

    return '\x1b';
  } else {
    return c;
  }
}

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

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

/*** append buffer ***/
```

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

compiles, but with no observable effects

Now let's make `Home` and `End` do something. For now, we'll have them move the cursor to the left or right edges of the screen.

kilo.c **Step 53** `home-end-simple`

```
/*** includes ***/
/*** defines ***/
/*** data ***/
/*** terminal ***/
/*** 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:
      {
        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

If you're on a laptop with an `Fn` key, you may be able to press `Fn`+`←` and `Fn`+`→` to simulate pressing the `Home` and `End` keys.

The `Delete` key

Lastly, let's detect when the `Delete` key is pressed. It simply sends the escape sequence `<esc>[3~`, so it's easy to add to our switch statement. We won't make this key do anything for now.

kilo.c **Step 54** `detect-delete-key`

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

#define KILO_VERSION "0.0.1"

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

enum editorKey {
  ARROW_LEFT = 1000,
  ARROW_RIGHT,
  ARROW_UP,
  ARROW_DOWN,
  DEL_KEY,
  HOME_KEY,
  END_KEY,
  PAGE_UP,
  PAGE_DOWN
};

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

void die(const char *s) { … }

void disableRawMode() { … }
```

```
void enableRawMode() { … }

int editorReadKey() {
  int nread;
  char c;
  while ((nread = read(STDIN_FILENO, &c, 1)) != 1) {
    if (nread == -1 && errno != EAGAIN) die("read");
  }

  if (c == '\x1b') {
    char seq[3];

    if (read(STDIN_FILENO, &seq[0], 1) != 1) return '\x1b';
    if (read(STDIN_FILENO, &seq[1], 1) != 1) return '\x1b';

    if (seq[0] == '[') {
      if (seq[1] >= '0' && seq[1] <= '9') {
        if (read(STDIN_FILENO, &seq[2], 1) != 1) return '\x1b';
        if (seq[2] == '~') {
          switch (seq[1]) {
            case '1': return HOME_KEY;
            case '3': return DEL_KEY;
            case '4': return END_KEY;
            case '5': return PAGE_UP;
            case '6': return PAGE_DOWN;
            case '7': return HOME_KEY;
            case '8': return END_KEY;
          }
        }
      } else {
        switch (seq[1]) {
          case 'A': return ARROW_UP;
          case 'B': return ARROW_DOWN;
          case 'C': return ARROW_RIGHT;
          case 'D': return ARROW_LEFT;
          case 'H': return HOME_KEY;
          case 'F': return END_KEY;
        }
      }
    } else if (seq[0] == 'O') {
      switch (seq[1]) {
        case 'H': return HOME_KEY;
        case 'F': return END_KEY;
      }
    }

    return '\x1b';
  } else {
    return c;
```

```
  }
}

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

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

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

compiles, but with no observable effects

If you're on a laptop with an `Fn` key, you may be able to press `Fn`+`Backspace` to simulate pressing the `Delete` key.

In the next chapter, we will get our program to display text files, complete with vertical and horizontal scrolling and a status bar.

`1.0.0beta11` (`changelog`)

top of page