

Build Your Own Text Editor

Welcome! This is an instruction booklet that shows you how to build a text editor in C.

The text editor is [antirez's kilo](#), with some changes. It's about 1000 lines of C in a single file with no dependencies, and it implements all the basic features you expect in a minimal editor, as well as syntax highlighting and a search feature.

This booklet walks you through building the editor in **184 steps**. Each step, you'll add, change, or remove a few lines of code. Most steps, you'll be able to **observe the changes** you made by compiling and running the program immediately afterwards.

I explain each step along the way, sometimes in a lot of detail. Feel free to skim or skip the prose, as the main point of this is that **you are going to build a text editor from scratch!** Anything you learn along the way is bonus, and there's plenty to learn just from typing in the changes to the code and observing the results.

See the [appendices](#) for more information on the tutorial itself (including what to do if you get stuck, and where to get help).

If you're ready to begin, then go to [chapter 1](#)!

Table of Contents

1. [Setup](#)
2. [Entering raw mode](#)
3. [Raw input and output](#)
4. [A text viewer](#)
5. [A text editor](#)
6. [Search](#)
7. [Syntax highlighting](#)
8. [Appendices](#)

[← back to snaptoken tutorials](#)

Setup



Ahh, step 1. Don't you love a fresh start on a blank slate? And then selecting that singular brick onto which you will build your entire palatial estate?

Unfortunately, when you're building a *computer program*, step 1 can get... complicated. And frustrating. You have to make sure your environment is set up for the programming language you're using, and you have to figure out how to compile and run your program in that environment.

Fortunately, the program we are building doesn't depend on any external libraries, so you don't need anything beyond a C compiler and the standard library it comes with. (We will also be using the `make` program.) To check whether you have a C compiler installed, try running `cc --version` at the command line (`cc` stands for "C Compiler"). To check whether you have `make`, try running `make -v`.

How to install a C compiler...

...in Windows

You will **need to install some kind of Linux environment within Windows**. This is because our text editor interacts with the terminal at a low level using the `<termios.h>` header, which isn't available on Windows. I suggest using either [Bash on Windows](#) or [Cygwin](#).

Bash on Windows: Only works on 64-bit Windows 10. See the [installation guide](#). After installing it, run `bash` at the command line whenever you want to enter the Linux

environment. Inside bash, run `sudo apt-get install gcc make` to install the GNU Compiler Collection and the make program. If `sudo` takes a really long time to do anything, you may have to [fix your /etc/hosts file](#).

Cygwin: Download the installer from cygwin.com/install.html. When the installer asks you to select packages to install, look in the `devel` category and select the `gcc-core` and `make` packages. To use Cygwin, you have to run the Cygwin terminal program. Unlike Bash on Windows, in Cygwin your home directory is separate from your Windows home directory. If you installed Cygwin to `C:\cygwin64`, then your home directory is at `C:\cygwin64\home\yourname`. So if you want to use a text editor outside of Cygwin to write your code, that's where you'll want to save to.

...in macOS

When you try to run the `cc` command, a window should pop up asking if you want to install the command line developer tools. You can also run `xcode-select --install` to get this window to pop up. Then just click “Install” and it will install a C compiler and `make`, among other things.

...in Linux

In Ubuntu, it's `sudo apt-get install gcc make`. Other distributions should have `gcc` and `make` packages available as well.

The `main()` function

Create a new file named `kilo.c` and give it a `main()` function. (`kilo` is the name of the text editor we are building.)

kilo.c

Step 1

main

```
int main() {  
    return 0;  
}
```

✔ compiles

In C, you have to put all your executable code inside functions. The `main()` function in C is special. It is the default starting point when you run your program. When you return from the `main()` function, the program exits and passes the returned integer back to the operating system. A return value of `0` indicates success.

C is a compiled language. That means we need to run our program through a C compiler to turn it into an executable file. We then run that executable like we would run any other program on the command line.

To compile `kilo.c`, run `cc kilo.c -o kilo` in your shell. If no errors occur, this will produce an executable named `kilo`. `-o` stands for “output”, and specifies that the output executable should be named `kilo`.

To run `kilo`, type `./kilo` in your shell and press `Enter`. The program doesn’t print any output, but you can check its exit status (the value `main()` returns) by running `echo $?`, which should print `0`.

Compiling with make

Typing `cc kilo.c -o kilo` every time you want to recompile gets tiring. The `make` program allows you to simply run `make` and it will compile your program for you. You just have to supply a `Makefile` to tell it how to compile your program.

Create a new file literally named `Makefile` with the following contents.

<u>Makefile</u>	Step 2	make
<pre>kilo: kilo.c \$(CC) kilo.c -o kilo -Wall -Wextra -pedantic -std=c99</pre>		
		 compiles

The first line says that `kilo` is what we want to build, and that `kilo.c` is what’s required to build it. The second line specifies the command to run in order to actually build `kilo` out of `kilo.c`. Make sure to indent the second line with an actual tab character, and not with spaces. You can indent C files however you want, but `Makefile`s must use tabs.

We have added a few things to the compilation command:

- `$(CC)` is a variable that `make` expands to `cc` by default.
- `-Wall` stands for “**all** Warnings”, and gets the compiler to warn you when it sees code in your program that might not technically be wrong, but is considered bad or questionable usage of the C language, like using variables before initializing them.
- `-Wextra` and `-pedantic` turn on even more warnings. For each step in this tutorial, if your program compiles, it shouldn’t produce any warnings except for “unused variable” warnings in some cases. If you get any other warnings, check to make sure your code exactly matches the code in that step.
- `-std=c99` specifies the exact version of the C language **standard** we’re using, which is [C99](#). C99 allows us to declare variables anywhere within a function, whereas [ANSI C](#) requires all variables to be declared at the top of a function or block.

Now that we have a `Makefile`, try running `make` to compile the program.

It may output `make: `kilo' is up to date..` It can tell that the current version of `kilo.c` has already been compiled by looking at each file's last-modified timestamp. If `kilo` was last modified after `kilo.c` was last modified, then `make` assumes that `kilo.c` has already been compiled, and so it doesn't bother running the compilation command. If `kilo.c` was last modified after `kilo` was, then `make` recompiles `kilo.c`. This is more useful for large projects with many different components to compile, as most of the components shouldn't need to be recompiled over and over when you're only making changes to one component's source code.

Try changing the return value in `kilo.c` to a number other than `0`. Then run `make`, and you should see it compile. Run `./kilo`, and try `echo $?` to see if you get the number you changed it to. Then change it back to `0`, recompile, and make sure it's back to returning `0`.

After each step in this tutorial, you will want to recompile `kilo.c`, see if it finds any errors in your code, and then run `./kilo`. It is easy to forget to recompile, and just run `./kilo`, and wonder why your changes to `kilo.c` don't seem to have any effect. You must recompile in order for changes in `kilo.c` to be reflected in `kilo`.

In the [next chapter](#), we'll work on getting the terminal into *raw mode*, and reading individual keypresses from the user.

Entering raw mode

Let's try and read keypresses from the user. (The lines you need to add are highlighted and marked with arrows.)

kilo.cStep 3read

```
#include <unistd.h>

int main() {
    char c;
    while (read(STDIN_FILENO, &c, 1) == 1);
    return 0;
}
```

↗ compiles

`read()` and `STDIN_FILENO` come from `<unistd.h>`. We are asking `read()` to read 1 byte from the standard input into the variable `c`, and to keep doing it until there are no more bytes to read. `read()` returns the number of bytes that it read, and will return `0` when it reaches the end of a file.

When you run `./kilo`, your terminal gets hooked up to the standard input, and so your keyboard input gets read into the `c` variable. However, by default your terminal starts in **canonical mode**, also called **cooked mode**. In this mode, keyboard input is only sent to your program when the user presses `Enter`. This is useful for many programs: it lets the user type in a line of text, use `Backspace` to fix errors until they get their input exactly the way they want it, and finally press `Enter` to send it to the program. But it does not work well for programs with more complex user interfaces, like text editors. We want to process each keypress as it comes in, so we can respond to it immediately.

What we want is **raw mode**. Unfortunately, there is no simple switch you can flip to set the terminal to raw mode. Raw mode is achieved by turning off a great many flags in the terminal, which we will do gradually over the course of this chapter.

To exit the above program, press `Ctrl-D` to tell `read()` that it's reached the end of file. Or you can always press `Ctrl-C` to signal the process to terminate immediately.

Press `q` to quit?

To demonstrate how canonical mode works, we'll have the program exit when it reads a `q` keypress from the user. (Lines you need to change are highlighted and marked the same way as lines you need to add.)

kilo.c

Step 4

press-q

```
#include <unistd.h>

int main() {
    char c;
    while (read(STDIN_FILENO, &c, 1) == 1 && c != 'q');
    return 0;
}
```

[↗ compiles](#)

To quit this program, you will have to type a line of text that includes a `q` in it, and then press enter. The program will quickly read the line of text one character at a time until it reads the `q`, at which point the `while` loop will stop and the program will exit. Any characters after the `q` will be left unread on the input queue, and you may see that input being fed into your shell after your program exits.

Turn off echoing

We can set a terminal's attributes by (1) using `tcgetattr()` to read the current attributes into a struct, (2) modifying the struct by hand, and (3) passing the modified struct to `tcsetattr()` to write the new terminal attributes back out. Let's try turning off the `ECHO` feature this way.

kilo.c

Step 5

echo

```
#include <termios.h>
#include <unistd.h>

void enableRawMode() {
    struct termios raw;

    tcgetattr(STDIN_FILENO, &raw);

    raw.c_lflag &= ~(ECHO);

    tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

int main() {
    enableRawMode();

    char c;
    while (read(STDIN_FILENO, &c, 1) == 1 && c != 'q');
    return 0;
}
```

[↗ compiles](#)

`struct termios`, `tcgetattr()`, `tcsetattr()`, `ECHO`, and `TCSAFLUSH` all come from `<termios.h>`.

The `ECHO` feature causes each key you type to be printed to the terminal, so you can see what you're typing. This is useful in canonical mode, but really gets in the way when we are trying to carefully render a user interface in raw mode. So we turn it off. This program does the same thing as the one in the previous step, it just doesn't print what you are typing. You may be familiar with this mode if you've ever had to type a password at the terminal, when using `sudo` for example.

After the program quits, depending on your shell, you may find your terminal is still not echoing what you type. Don't worry, it will still listen to what you type. Just press `ctrl-c` to start a fresh line of input to your shell, and type in `reset` and press `Enter`. This resets your terminal back to normal in most cases. Failing that, you can always restart your terminal emulator. We'll fix this whole problem in the next step.

Terminal attributes can be read into a `termios` struct by `tcgetattr()`. After modifying them, you can then apply them to the terminal using `tcsetattr()`. The `TCSAFLUSH` argument specifies when to apply the change: in this case, it waits for all pending output to be written to the terminal, and also discards any input that hasn't been read.

The `c_lflag` field is for “local flags”. A comment in macOS’s `<termios.h>` describes it as a “dumping ground for other state”. So perhaps it should be thought of as “miscellaneous flags”. The other flag fields are `c_iflag` (input flags), `c_oflag` (output flags), and `c_cflag` (control flags), all of which we will have to modify to enable raw mode.

ECHO is a [bitflag](#), defined as 000000000000000000000000000000001000 in binary. We use the bitwise-NOT operator (~) on this value to get 111111111111111111111111111110111. We then bitwise-AND this value with the flags field, which forces the fourth bit in the flags field to become 0, and causes every other bit to retain its current value. Flipping bits like this is common in C.

Disable raw mode at exit

Let's be nice to the user and restore their terminal's original attributes when our program exits. We'll save a copy of the `termios` struct in its original state, and use `tcsetattr()` to apply it to the terminal when the program exits.

kilo.c

Step 6

```
atexit
```

```
#include <stdlib.h>
#include <termios.h>
#include <unistd.h>

struct termios orig_termios;
```

```

void disableRawMode() {
    tcsetattr(STDIN_FILENO, TCSAFLUSH, &orig_termios);
}

void enableRawMode() {
    tcgetattr(STDIN_FILENO, &orig_termios);
    atexit(disableRawMode);

    struct termios raw = orig_termios;
    raw.c_lflag &= ~(ECHO);

    tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

int main() { ... }

```

↗ compiles

`atexit()` comes from `<stdlib.h>`. We use it to register our `disableRawMode()` function to be called automatically when the program exits, whether it exits by returning from `main()`, or by calling the `exit()` function. This way we can ensure we'll leave the terminal attributes the way we found them when our program exits.

We store the original terminal attributes in a global variable, `orig_termios`. We assign the `orig_termios` struct to the `raw` struct in order to make a copy of it before we start making our changes.

You may notice that leftover input is no longer fed into your shell after the program quits. This is because of the `TCSAFLUSH` option being passed to `tcsetattr()` when the program exits. As described earlier, it discards any unread input before applying the changes to the terminal. (Note: This doesn't happen in Cygwin for some reason, but it won't matter once we are reading input one byte at a time.)

Turn off canonical mode

There is an `ICANON` flag that allows us to turn off canonical mode. This means we will finally be reading input byte-by-byte, instead of line-by-line.

kilo.c

Step 7

icanon

```

#include <stdlib.h>
#include <termios.h>
#include <unistd.h>

struct termios orig_termios;

void disableRawMode() { ... }

void enableRawMode() {
    tcgetattr(STDIN_FILENO, &orig_termios);

```

```

atexit(disableRawMode);

struct termios raw = orig_termios;
raw.c_lflag &= ~(ECHO | ICANON);

tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

int main() { ... }

```

[↗ compiles](#)

ICANON comes from `<termios.h>`. Input flags (the ones in the `c_iflag` field) generally start with I like ICANON does. However, ICANON is not an input flag, it's a "local" flag in the `c_lflag` field. So that's confusing.

Now the program will quit as soon as you press `q`].

Display keypresses

To get a better idea of how input in raw mode works, let's print out each byte that we `read()`. We'll print each character's numeric ASCII value, as well as the character it represents if it is a printable character.

kilo.c

Step 8

keypresses

```

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

struct termios orig_termios;

void disableRawMode() { ... }

void enableRawMode() { ... }

int main() {
    enableRawMode();

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

    return 0;
}

```

`isctrl()` comes from `<ctype.h>`, and `printf()` comes from `<stdio.h>`.

`isctrl()` tests whether a character is a control character. Control characters are nonprintable characters that we don't want to print to the screen. ASCII codes 0–31 are all control characters, and 127 is also a control character. ASCII codes 32–126 are all printable. (Check out the [ASCII table](#) to see all of the characters.)

`printf()` can print multiple representations of a byte. `%d` tells it to format the byte as a decimal number (its ASCII code), and `%c` tells it to write out the byte directly, as a character.

This is a very useful program. It shows us how various keypresses translate into the bytes we read. Most ordinary keys translate directly into the characters they represent. But try seeing what happens when you press the arrow keys, or `Escape`, or `Page Up`, or `Page Down`, or `Home`, or `End`, or `Backspace`, or `Delete`, or `Enter`. Try key combinations with `ctrl`, like `ctrl-A`, `ctrl-B`, etc.

You'll notice a few interesting things:

- Arrow keys, `Page Up`, `Page Down`, `Home`, and `End` all input 3 or 4 bytes to the terminal: `27`, `'['`, and then one or two other characters. This is known as an *escape sequence*. All escape sequences start with a `27` byte. Pressing `Escape` sends a single `27` byte as input.
- `Backspace` is byte `127`. `Delete` is a 4-byte escape sequence.
- `Enter` is byte `10`, which is a newline character, also known as `'\n'`.
- `ctrl-A` is `1`, `ctrl-B` is `2`, `ctrl-C` is... oh, that terminates the program, right. But the `ctrl` key combinations that do work seem to map the letters A–Z to the codes 1–26.

By the way, if you happen to press `ctrl-S`, you may find your program seems to be frozen. What you've done is you've asked your program to [stop sending you output](#). Press `ctrl-Q` to tell it to resume sending you output.

Also, if you press `ctrl-Z` (or maybe `ctrl-Y`), your program will be suspended to the background. Run the `fg` command to bring it back to the foreground. (It may quit immediately after you do that, as a result of `read()` returning `-1` to indicate that an error occurred. This happens on macOS, while Linux seems to be able to resume the `read()` call properly.)

Turn off `ctrl-C` and `ctrl-Z` signals

By default, `ctrl-C` sends a SIGINT signal to the current process which causes it to terminate, and `ctrl-Z` sends a SIGTSTP signal to the current process which causes it to suspend. Let's turn off the sending of both of these signals.

kilo.cStep 9isig

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

struct termios orig_termios;

void disableRawMode() { ... }

void enableRawMode() {
    tcgetattr(STDIN_FILENO, &orig_termios);
    atexit(disableRawMode);

    struct termios raw = orig_termios;
    raw.c_lflag &= ~(ECHO | ICANON | ISIG);

    tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

int main() { ... }
```

✖ compiles

ISIG comes from `<termios.h>`. Like ICANON, it starts with I but isn't an input flag.

Now `ctrl-C` can be read as a 3 byte and `ctrl-Z` can be read as a 26 byte.

This also disables `ctrl-Y` on macOS, which is like `ctrl-Z` except it waits for the program to read input before suspending it.

Disable `Ctrl-S` and `Ctrl-Q`

By default, `ctrl-S` and `ctrl-Q` are used for [software flow control](#). `ctrl-S` stops data from being transmitted to the terminal until you press `ctrl-Q`. This originates in the days when you might want to pause the transmission of data to let a device like a printer catch up. Let's just turn off that feature.

kilo.cStep 10ixon

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
```

```

#include <unistd.h>

struct termios orig_termios;

void disableRawMode() { ... }

void enableRawMode() {
    tcgetattr(STDIN_FILENO, &orig_termios);
    atexit(disableRawMode);

    struct termios raw = orig_termios;
    raw.c_iflag &= ~(IXON);
    raw.c_lflag &= ~(ECHO | ICANON | ISIG);

    tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

int main() { ... }

```

✖ compiles

IXON comes from <termios.h>. The I stands for “input flag” (which it is, unlike the other I flags we’ve seen so far) and XON comes from the names of the two control characters that `ctrl-s` and `ctrl-q` produce: XOFF to pause transmission and XON to resume transmission.

Now `ctrl-s` can be read as a 19 byte and `ctrl-q` can be read as a 17 byte.

Disable `Ctrl-V`

On some systems, when you type `ctrl-v`, the terminal waits for you to type another character and then sends that character literally. For example, before we disabled `ctrl-c`, you might’ve been able to type `ctrl-v` and then `ctrl-c` to input a 3 byte. We can turn off this feature using the IEXTEN flag.

Turning off IEXTEN also fixes `ctrl-o` in macOS, whose terminal driver is otherwise set to discard that control character.

kilo.c

Step 11

iexten

```

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

struct termios orig_termios;

void disableRawMode() { ... }

void enableRawMode() {

```

```

tcgetattr(STDIN_FILENO, &orig_termios);
atexit(disableRawMode);

struct termios raw = orig_termios;
raw.c_iflag &= ~(IXON);
raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

int main() { ... }

```

[↗ compiles](#)

IEXTEN comes from `<termios.h>`. It is another flag that starts with I but belongs in the `c_lflag` field.

`ctrl-v` can now be read as a 22 byte, and `ctrl-o` as a 15 byte.

Fix Ctrl-M

If you run the program now and go through the whole alphabet while holding down `ctrl`, you should see that we have every letter except `M`. `ctrl-M` is weird: it's being read as 10, when we expect it to be read as 13, since it is the 13th letter of the alphabet, and `ctrl-j` already produces a 10. What else produces 10? The `Enter` key does.

It turns out that the terminal is helpfully translating any carriage returns (13, `'\r'`) inputted by the user into newlines (10, `'\n'`). Let's turn off this feature.

kilo.c

Step 12

icrnl

```

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

struct termios orig_termios;

void disableRawMode() { ... }

void enableRawMode() {
    tcgetattr(STDIN_FILENO, &orig_termios);
    atexit(disableRawMode);

    struct termios raw = orig_termios;
    raw.c_iflag &= ~(ICRNL | IXON);
    raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

    tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

```

```
}  
  
int main() { ... }
```

↗ compiles

ICRNL comes from `<termios.h>`. The I stands for “input flag”, CR stands for “carriage return”, and NL stands for “new line”.

Now `ctrl-M` is read as a 13 (carriage return), and the `Enter` key is also read as a 13.

Turn off all output processing

It turns out that the terminal does a similar translation on the output side. It translates each newline (`"\n"`) we print into a carriage return followed by a newline (`"\r\n"`). The terminal requires both of these characters in order to start a new line of text. The carriage return moves the cursor back to the beginning of the current line, and the newline moves the cursor down a line, scrolling the screen if necessary. (These two distinct operations originated in the days of typewriters and [teletypes](#).)

We will turn off all output processing features by turning off the `OPOST` flag. In practice, the `"\n"` to `"\r\n"` translation is likely the only output processing feature turned on by default.

kilo.c

Step 13

opost

```
#include <ctype.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <termios.h>  
#include <unistd.h>  
  
struct termios orig_termios;  
  
void disableRawMode() { ... }  
  
void enableRawMode() {  
    tcgetattr(STDIN_FILENO, &orig_termios);  
    atexit(disableRawMode);  
  
    struct termios raw = orig_termios;  
    raw.c_iflag &= ~(ICRNL | IXON);  
    raw.c_oflag &= ~(OPOST);  
    raw.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);  
  
    tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);  
}  
  
int main() { ... }
```

↗ compiles

OPOST comes from `<termios.h>`. `O` means it's an output flag, and I assume `POST` stands for “post-processing of output”.

If you run the program now, you'll see that the newline characters we're printing are only moving the cursor down, and not to the left side of the screen. To fix that, let's add carriage returns to our `printf()` statements.

<u>kilo.c</u>	Step 14	carriage-returns
<pre>#include <ctype.h> #include <stdio.h> #include <stdlib.h> #include <termios.h> #include <unistd.h> struct termios orig_termios; void disableRawMode() { ... } void enableRawMode() { ... } int main() { enableRawMode(); char c; while (read(STDIN_FILENO, &c, 1) == 1 && c != 'q') { if (iscntrl(c)) { printf("%d\r\n", c); } else { printf("%d ('%c')\r\n", c, c); } } return 0; }</pre>		
		↗ compiles

From now on, we'll have to write out the full `"\r\n"` whenever we want to start a new line.

Miscellaneous flags

Let's turn off a few more flags.

<u>kilo.c</u>	Step 15	misc-flags
<pre>#include <ctype.h> #include <stdio.h> #include <stdlib.h> #include <termios.h> #include <unistd.h></pre>		

```

struct termios orig_termios;

void disableRawMode() { ... }

void enableRawMode() {
    tcgetattr(STDIN_FILENO, &orig_termios);
    atexit(disableRawMode);

    struct termios raw = 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);

    tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

int main() { ... }

```

[↪](#) compiles, but with no observable effects

BRKINT, INPCK, ISTRIP, and CS8 all come from `<termios.h>`.

This step probably won't have any observable effect for you, because these flags are either already turned off, or they don't really apply to modern terminal emulators. But at one time or another, switching them off was considered (by someone) to be part of enabling "raw mode", so we carry on the tradition (of whoever that someone was) in our program.

As far as I can tell:

- When BRKINT is turned on, a [break condition](#) will cause a SIGINT signal to be sent to the program, like pressing `Ctrl-C`.
- INPCK enables parity checking, which doesn't seem to apply to modern terminal emulators.
- ISTRIP causes the 8th bit of each input byte to be stripped, meaning it will set it to 0. This is probably already turned off.
- CS8 is not a flag, it is a bit mask with multiple bits, which we set using the bitwise-OR (`|`) operator unlike all the flags we are turning off. It sets the character size (CS) to 8 bits per byte. On my system, it's already set that way.

A timeout for `read()`

Currently, `read()` will wait indefinitely for input from the keyboard before it returns. What if we want to do something like animate something on the screen while waiting for user input? We can set a timeout, so that `read()` returns if it doesn't get any input for a certain amount of time.

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

struct termios orig_termios;

void disableRawMode() { ... }

void enableRawMode() {
    tcgetattr(STDIN_FILENO, &orig_termios);
    atexit(disableRawMode);

    struct termios raw = 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;

    tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw);
}

int main() {
    enableRawMode();

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

    return 0;
}
```

[↗ compiles](#)

VMIN and VTIME come from `<termios.h>`. They are indexes into the `c_cc` field, which stands for “control characters”, an array of bytes that control various terminal settings.

The VMIN value sets the minimum number of bytes of input needed before `read()` can return. We set it to 0 so that `read()` returns as soon as there is any input to be read. The VTIME value sets the maximum amount of time to wait before `read()`

returns. It is in tenths of a second, so we set it to 1/10 of a second, or 100 milliseconds. If `read()` times out, it will return 0, which makes sense because its usual return value is the number of bytes read.

When you run the program, you can see how often `read()` times out. If you don't supply any input, `read()` returns without setting the `c` variable, which retains its 0 value and so you see 0s getting printed out. If you type really fast, you can see that `read()` returns right away after each keypress, so it's not like you can only read one keypress every tenth of a second.

If you're using **Bash on Windows**, you may see that `read()` still blocks for input. It doesn't seem to care about the `VTIME` value. Fortunately, this won't make too big a difference in our text editor, as we'll be basically blocking for input anyways.

Error handling

`enableRawMode()` now gets us fully into raw mode. It's time to clean up the code by adding some error handling.

First, we'll add a `die()` function that prints an error message and exits the program.

<u>kilo.c</u>	Step 17	die
<pre>#include <ctype.h> #include <stdio.h> #include <stdlib.h> #include <termios.h> #include <unistd.h> struct termios orig_termios; void die(const char *s) { perror(s); exit(1); } void disableRawMode() { ... } void enableRawMode() { ... } int main() { ... }</pre>		
↗ compiles, but with no observable effects		

`perror()` comes from `<stdio.h>`, and `exit()` comes from `<stdlib.h>`.

Most C library functions that fail will set the global `errno` variable to indicate what the error was. `perror()` looks at the global `errno` variable and prints a descriptive error message for it. It also prints the string given to it before it prints the error

message, which is meant to provide context about what part of your code caused the error.

After printing out the error message, we exit the program with an exit status of 1, which indicates failure (as would any non-zero value).

Let's check each of our library calls for failure, and call `die()` when they fail.

kilo.c

Step 18

error-handling

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

struct termios orig_termios;

void die(const char *s) { ... }

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

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

    struct termios raw = 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");
}

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 == 'q') break;
}

return 0;
}

```

[↗ compiles](#)

`errno` and `EAGAIN` come from `<errno.h>`.

`tcsetattr()`, `tcgetattr()`, and `read()` all return `-1` on failure, and set the `errno` value to indicate the error.

In Cygwin, when `read()` times out it returns `-1` with an `errno` of `EAGAIN`, instead of just returning `0` like it's supposed to. To make it work in Cygwin, we won't treat `EAGAIN` as an error.

An easy way to make `tcgetattr()` fail is to give your program a text file or a pipe as the standard input instead of your terminal. To give it a file as standard input, run `./kilo <kilo.c`. To give it a pipe, run `echo test | ./kilo`. Both should result in the same error from `tcgetattr()`, something like `Inappropriate ioctl for device`.

Sections

That just about concludes this chapter on entering raw mode. The last thing we'll do now is split our code into sections. This will allow these diffs to be shorter, as each section that isn't changed in a diff will be folded into a single line.

kilo.c

Step 19

sections

```

/** includes */

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

/** data */

struct termios orig_termios;

/** terminal */

void die(const char *s) { ... }

void disableRawMode() { ... }

```

```
void enableRawMode() { ... }
```

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

```
int main() { ... }
```

[↗](#) compiles, but with no observable effects

In the [next chapter](#), we'll do some more low-level terminal input/output handling, and use that to draw to the screen and allow the user to move the cursor around.

[1.0.0beta11](#) ([changelog](#))

[top of page](#)

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.

```
/** 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.

```
/** 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 **IO**ctl (which itself stands for Input/Output Control) **G**et **W**indow **S**ize.)

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	<code>screenrows</code>
---------------	----------------	-------------------------

```

/** 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.cStep 30bottom-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.

<u>kilo.c</u>	Step 36	abuf-struct
<pre>/** 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 */</pre>		
		↗ 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
<pre>/** 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 */</pre>		
↗ 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.cStep 41welcome

```
/** 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.

```

/** 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.

```
/** 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`.

```
/** 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` (this is the letter O followed by H). Similarly, the `End` key could be sent as `<esc>[4~`, `<esc>[8~`, `<esc>[F`, or `<esc>OF` (this is the letter O followed by F). 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 */
```

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](#)

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-- > 0) 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
<pre>/** 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 */</pre>		
		 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-- > 0) 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[]) { ... }
```

⌘ 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 */`.

```
/** 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;
    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 */
```

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.

kilo.c	Step 63	fix-append-row
<pre> **** 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 ****/ </pre>		
 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
<pre> **** includes ****/ **** defines ****/ **** data ****/ **** terminal ****/ **** row operations ****/ **** file i/o ****/ **** append buffer ****/ </pre>		

```

/** 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-- > 0) 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`.

<u>kilo.c</u>	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.

kilo.c

Step 68

editor-scroll

```

/** 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 row 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-- > 0) 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 */

```

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 `rsiz` (which contains the size of the contents of `render`) to the `erow` struct, and initializing them in `editorAppendRow()`, which is where new `erows` get constructed and initialized.

kilo.c

Step 80

render

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

typedef struct erow {
    int size;
    int rsiz;
    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 row.

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->rsz = 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
<pre>/** 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 */</pre>		
↪ 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
<pre>/** includes */ /** defines */ /** data */ /** terminal */</pre>		

```

/** 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.

kilo.c

Step 90

page-up-down

```

/** 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


```

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>
#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 `vsprintf()` 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.

[1.0.0beta11](#) ([changelog](#))

[top of page](#)

A text editor

Insert ordinary characters

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

kilo.c**Step 101**

row-insert-char

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

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

void editorUpdateRow(erow *row) { ... }

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

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

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

[↗](#) compiles, but with no observable effects

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

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

Then we allocate one more byte for the `chars` of the `erow` (we add 2 because we also have to make room for the null byte), and use `memmove()` to make room for the new character. We increment the `size` of the `chars` array, and then actually assign the

character to its position in the array. Finally, we call `editorUpdateRow()` so that the `render` and `rsize` fields get updated with the new row content.

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

kilo.c	Step 102	editor-insert-char
<pre>/** includes */ /** defines */ /** data */ /** terminal */ /** row operations */ int editorRowCxToRx(erow *row, int cx) { ... } void editorUpdateRow(erow *row) { ... } void editorAppendRow(char *s, size_t len) { ... } void editorRowInsertChar(erow *row, int at, int c) { ... } /** editor operations */ void editorInsertChar(int c) { if (E.cy == E.numrows) { editorAppendRow("", 0); } editorRowInsertChar(&E.row[E.cy], E.cx, c); E.cx++; } /** file i/o */ /** append buffer */ /** output */ /** input */ /** init */</pre>		
		↗ compiles, but with no observable effects

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

Notice that `editorInsertChar()` doesn't have to worry about the details of modifying an `erow`, and `editorRowInsertChar()` doesn't have to worry about

where the cursor is. That is the difference between functions in the `/** editor operations */` section and functions in the `/** row operations */` section.

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

kilo.c

Step 103

key-insert-char

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

void editorMoveCursor(int key) { ... }

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

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

        case HOME_KEY:
            E.cx = 0;
            break;

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

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

```

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

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

default:
    editorInsertChar(c);
    break;
}
}

/**** init ****/

```

[↗ compiles](#)

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

Prevent inserting special characters

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

kilo.c

Step 104

block-special-chars

```

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

#define KILO_VERSION "0.0.1"
#define KILO_TAB_STOP 8

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

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

```

```

PAGE_DOWN
};

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

void editorMoveCursor(int key) { ... }

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

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

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

        case HOME_KEY:
            E.cx = 0;
            break;

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

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

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

```

```

    int times = E.screenrows;
    while (times-->0)
        editorMoveCursor(c == PAGE_UP ? ARROW_UP : ARROW_DOWN);
    }
    break;

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

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

default:
    editorInsertChar(c);
    break;
}
}

/** init */

```

 compiles

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

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

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

Lastly, we handle `ctrl-L` and `Escape` by not doing anything when those keys are pressed. `ctrl-L` is traditionally used to refresh the screen in terminal programs. In our text editor, the screen refreshes after *any* keypress, so we don't have to do anything else to implement that feature. We ignore the `Escape` key because there are many key escape sequences that we aren't handling (such as the `F1`–`F12` keys), and the way we wrote `editorReadKey()`, pressing those keys will be equivalent to pressing the

`Escape` key. We don't want the user to unwittingly insert the escape character `27` into their text, so we ignore those keypresses.

Save to disk

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

kilo.c

Step 105

rows-to-string

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

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

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

    return buf;
}

void editorOpen(char *filename) { ... }
```

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

[↵](#) compiles, but with no observable effects

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

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

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

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

kilo.c

Step 106

save

```
/** includes */

#define _DEFAULT_SOURCE
#define _BSD_SOURCE
#define _GNU_SOURCE

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

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

char *editorRowsToString(int *buflen) { ... }

void editorOpen(char *filename) { ... }

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

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

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

```

}

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

```

[↪](#) compiles, but with no observable effects

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

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

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

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

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

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

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

kilo.c

Step 107

ctrl-s

```

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

```

```

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

void editorMoveCursor(int key) { ... }

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

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

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

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

        case HOME_KEY:
            E.cx = 0;
            break;

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

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

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

```

```

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

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

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

default:
    editorInsertChar(c);
    break;
}
}

/**/ init /**/

```

[↗ compiles](#)

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

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

kilo.c

Step 108

save-errors

```

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

char *editorRowsToString(int *buflen) { ... }

void editorOpen(char *filename) { ... }

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

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

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

```

```

    if (ftruncate(fd, len) != -1) {
        if (write(fd, buf, len) == len) {
            close(fd);
            free(buf);
            return;
        }
    }
    close(fd);
}

free(buf);
}

```

```

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

```

[↗ compiles](#)

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

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

kilo.c

Step 109

save-status-message

```

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

char *editorRowsToString(int *buflen) { ... }

void editorOpen(char *filename) { ... }

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

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

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

```

```

        free(buf);
        editorSetStatusMessage("%d bytes written to disk", len);
        return;
    }
}
close(fd);
}

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

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

void initEditor() { ... }

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

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

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

    return 0;
}

```

⌘ doesn't compile

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

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

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

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

<u>kilo.c</u>	Step 110	prototypes
<pre>/** includes */ /** defines */ /** data */ typedef struct erow { ... } erow; struct editorConfig { ... }; struct editorConfig E; /** prototypes */ void editorSetStatusMessage(const char *fmt, ...); /** terminal */ /** row operations */ /** editor operations */ /** file i/o */ /** append buffer */ /** output */ /** input */ /** init */</pre>		
		↗ compiles

Dirty flag

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

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

<u>kilo.c</u>	Step 111	dirty
<pre>/** includes */ /** defines */ /** data */ typedef struct erow { ... } erow; struct editorConfig { int cx, cy;</pre>		

```

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

```

```

struct editorConfig E;

```

```

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

```

```

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

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

```

```

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

```

[↪](#) compiles, but with no observable effects

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

kilo.c

Step 112

show-dirty

```

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

```

```

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

void editorScroll() { ... }

void editorDrawRows(struct abuf *ab) { ... }

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

void editorDrawMessageBar(struct abuf *ab) { ... }

void editorRefreshScreen() { ... }

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

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

```

[↪](#) compiles, but with no observable effects

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

kilo.c

Step 113

increment-dirty

```

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

```

```

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

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

void editorUpdateRow(erow *row) { ... }

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

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

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

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

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

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

```

 compiles

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

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

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

char *editorRowsToString(int *buflen) { ... }

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

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

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

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

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

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

```

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

```

```

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

```

[↗ compiles](#)

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

Quit confirmation

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

kilo.c

Step 115

quit-confirmation

```

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

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

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

enum editorKey { ... };

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

void editorMoveCursor(int key) { ... }

void editorProcessKeypress() {
    static int quit_times = KILO_QUIT_TIMES;

    int c = editorReadKey();

    switch (c) {
        case '\r':

```

```

    /* TODO */
    break;

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

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

case HOME_KEY:
    E.cx = 0;
    break;

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

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

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

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

case ARROW_UP:
case ARROW_DOWN:
case ARROW_LEFT:

```

```

    case ARROW_RIGHT:
        editorMoveCursor(c);
        break;

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

    default:
        editorInsertChar(c);
        break;
}

quit_times = KILO_QUIT_TIMES;
}

/** init */

```

↗ compiles

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

Simple backspacing

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

kilo.c

Step 116

row-del-char

```

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

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

void editorUpdateRow(erow *row) { ... }

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

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

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

```

```

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

```

```

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

```

[↗](#) compiles, but with no observable effects

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

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

kilo.c

Step 117

editor-del-char

```

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

void editorInsertChar(int c) { ... }

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

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

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

```

[↗](#) compiles, but with no observable effects

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

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

kilo.c

Step 118

key-del-char

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

void editorMoveCursor(int key) { ... }

void editorProcessKeypress() {
    static int quit_times = KILO_QUIT_TIMES;

    int c = editorReadKey();

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

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

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

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

```

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

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

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

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

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

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

default:
    editorInsertChar(c);
    break;
}

quit_times = KILO_QUIT_TIMES;
}

/* init */

```

[↗ compiles](#)

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

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

Backspacing at the start of a line

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

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

kilo.c

Step 119

del-row

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

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

void editorUpdateRow(erow *row) { ... }

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

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

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

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

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

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

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

[↗](#) compiles, but with no observable effects

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

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

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

kilo.c

Step 120

row-append-string

```
/**/ includes ***/  
/**/ defines ***/  
/**/ data ***/  
/**/ prototypes ***/  
/**/ terminal ***/  
/**/ row operations ***/  
  
int editorRowCxToRx(erow *row, int cx) { ... }  
  
void editorUpdateRow(erow *row) { ... }  
  
void editorAppendRow(char *s, size_t len) { ... }  
  
void editorFreeRow(erow *row) { ... }  
  
void editorDelRow(int at) { ... }  
  
void editorRowInsertChar(erow *row, int at, int c) { ... }  
  
void editorRowAppendString(erow *row, char *s, size_t len) {  
    row->chars = realloc(row->chars, row->size + len + 1);  
    memcpy(&row->chars[row->size], s, len);  
    row->size += len;  
    row->chars[row->size] = '\0';  
    editorUpdateRow(row);  
    E.dirty++;  
}  
  
void editorRowDelChar(erow *row, int at) { ... }  
  
/**/ editor operations ***/  
/**/ file i/o ***/  
/**/ append buffer ***/  
/**/ output ***/
```

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

[↗](#) compiles, but with no observable effects

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

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

kilo.c

Step 121

del-char-row

```
/**/ includes ***/  
/**/ defines ***/  
/**/ data ***/  
/**/ prototypes ***/  
/**/ terminal ***/  
/**/ row operations ***/  
/**/ editor operations ***/  
  
void editorInsertChar(int c) { ... }  
  
void editorDelChar() {  
    if (E.cy == E.numrows) return;  
    if (E.cx == 0 && E.cy == 0) return;  
  
    erow *row = &E.row[E.cy];  
    if (E.cx > 0) {  
        editorRowDelChar(row, E.cx - 1);  
        E.cx--;  
    } else {  
        E.cx = E.row[E.cy - 1].size;  
        editorRowAppendString(&E.row[E.cy - 1], row->chars, row->size);  
        editorDelRow(E.cy);  
        E.cy--;  
    }  
}  
  
/**/ file i/o ***/  
/**/ append buffer ***/  
/**/ output ***/  
/**/ input ***/  
/**/ init ***/
```

[↗](#) compiles

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

then delete the row that `E.cx` is on. We set `E.cx` to the end of the contents of the previous row before appending to that row. That way, the cursor will end up at the point where the two lines joined.

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

The `Enter` key

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

kilo.c

Step 122

append-to-insert

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

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

void editorUpdateRow(erow *row) { ... }

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

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

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

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

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

```

void editorFreeRow(erow *row) { ... }

void editorDelRow(int at) { ... }

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

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

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

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

```

⌘ doesn't compile

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

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

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

kilo.c

Step 123

use-insert-row

```

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

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

void editorDelChar() { ... }

/** file i/o */

char *editorRowsToString(int *buflen) { ... }

```

```

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

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

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

void editorSave() { ... }

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

```

[↗](#) compiles, but with no observable effects

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

kilo.c

Step 124

insert-newline

```

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

void editorInsertChar(int c) { ... }

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

```

```

        editorUpdateRow(row);
    }
    E.cy++;
    E.cx = 0;
}

void editorDelChar() { ... }

```

```

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

```

[↪](#) compiles, but with no observable effects

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

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

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

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

kilo.c

Step 125

enter-key

```

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

void editorMoveCursor(int key) { ... }

void editorProcessKeypress() {
    static int quit_times = KILO_QUIT_TIMES;

```

```

int c = editorReadKey();

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

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

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

    case HOME_KEY:
        E.cx = 0;
        break;

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

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

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

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

```

```

        break;

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

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

    default:
        editorInsertChar(c);
        break;
}

quit_times = KILO_QUIT_TIMES;
}

/**** init ****/

```

[↗ compiles](#)

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

Save as...

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

kilo.c

Step 126

prompt

```

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

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

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

```

```

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

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

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

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

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

void editorMoveCursor(int key) { ... }

void editorProcessKeypress() { ... }

/**** init ****/

```

[↗](#) compiles, but with no observable effects

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

When the user presses `Enter`, and their input is not empty, the status message is cleared and their input is returned. Otherwise, when they input a printable character, we append it to `buf`. If `buflen` has reached the maximum capacity we allocated (stored in `bufsize`), then we double `bufsize` and allocate that amount of memory before appending to `buf`. We also make sure that `buf` ends with a `\0` character,

because both `editorSetStatusMessage()` and the caller of `editorPrompt()` will use it to know where the string ends.

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

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

kilo.c

Step 127

save-as

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

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

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

char *editorRowsToString(int *buflen) { ... }

void editorOpen(char *filename) { ... }

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

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

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

```

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

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

```

[↗ compiles](#)

Great, we now have basic “Save as...” functionality. Next, let’s allow the user to press Escape to cancel the input prompt.

kilo.c

Step 128

prompt-escape

```

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

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

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

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

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

```

```

        buf[buflen++] = c;
        buf[buflen] = '\0';
    }
}
}

void editorMoveCursor(int key) { ... }

void editorProcessKeypress() { ... }

/* init */

```

[↗ compiles](#)

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

kilo.c

Step 129

abort-save

```

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

char *editorRowsToString(int *buflen) { ... }

void editorOpen(char *filename) { ... }

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

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

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

```

```

    }
}
close(fd);
}

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

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

```

[↗ compiles](#)

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

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

kilo.c

Step 130

prompt-backspace

```

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

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

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

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

        int c = editorReadKey();
        if (c == DEL_KEY || c == CTRL_KEY('h') || c == BACKSPACE) {
            if (buflen != 0) buf[--buflen] = '\0';
        } else if (c == '\x1b') {
            editorSetStatusMessage("");

```

```

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

void editorMoveCursor(int key) { ... }

void editorProcessKeypress() { ... }

/**/ init ***/

```

[↗ compiles](#)

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

[1.0.0beta11](#) ([changelog](#))

[top of page](#)

Search

Let's use `editorPrompt()` to implement a minimal search feature. When the user types a search query and presses `Enter`, we'll loop through all the rows of the file, and if a row contains their query string, we'll move the cursor to the match.

kilo.c**Step 131**

basic-search

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

char *editorRowsToString(int *buflen) { ... }

void editorOpen(char *filename) { ... }

void editorSave() { ... }

/** find */

void editorFind() {
    char *query = editorPrompt("Search: %s (ESC to cancel)");
    if (query == NULL) return;

    int i;
    for (i = 0; i < E.numrows; i++) {
        erow *row = &E.row[i];
        char *match = strstr(row->render, query);
        if (match) {
            E.cy = i;
            E.cx = match - row->render;
            E.rowoff = E.numrows;
            break;
        }
    }

    free(query);
}

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

[↪](#) compiles, but with no observable effects

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

If they pressed `Escape` to cancel the input prompt, then `editorPrompt()` returns `NULL` and we abort the search.

Otherwise, we loop through all the rows of the file. We use `strstr()` to check if `query` is a substring of the current row. It returns `NULL` if there is no match, otherwise it returns a pointer to the matching substring. To convert that into an index that we can set `E.cx` to, we subtract the `row->render` pointer from the `match` pointer, since `match` is a pointer into the `row->render` string. Lastly, we set `E.rowoff` so that we are scrolled to the very bottom of the file, which will cause `editorScroll()` to scroll upwards at the next screen refresh so that the matching line will be at the very top of the screen. This way, the user doesn't have to look all over their screen to find where their cursor jumped to, and where the matching line is.

There's one problem here. Did you notice what we just did wrong? We assigned a `render` index to `E.cx`, but `E.cx` is an index into `chars`. If there are tabs to the left of the match, the cursor is going to be in the wrong position. We need to convert the `render` index into a `chars` index before assigning it to `E.cx`. Let's create an `editorRowRxToCx()` function, which is the opposite of the `editorRowCxToRx()` function we wrote in [chapter 4](#), but contains a lot of the same code.

kilo.c

Step 132

rx-to-cx

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

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

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

        if (cur_rx > rx) return cx;
    }
    return cx;
}

void editorUpdateRow(erow *row) { ... }

void editorInsertRow(int at, char *s, size_t len) { ... }
```

```

void editorFreeRow(erow *row) { ... }

void editorDelRow(int at) { ... }

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

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

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

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

```

[↪](#) compiles, but with no observable effects

To convert an `rx` into a `cx`, we do pretty much the same thing when converting the other way: loop through the `chars` string, calculating the current `rx` value (`cur_rx`) as we go. But instead of stopping when we hit a particular `cx` value and returning `cur_rx`, we want to stop when `cur_rx` hits the given `rx` value and return `cx`.

The `return` statement at the very end is just in case the caller provided an `rx` that's out of range, which shouldn't happen. The `return` statement inside the `for` loop should handle all `rx` values that are valid indexes into `render`.

Now let's call `editorRowRxToCx()` to convert the matched index to a `chars` index and assign that to `E.cx`.

kilo.c

Step 133

use-rx-to-cx

```

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

void editorFind() {
    char *query = editorPrompt("Search: %s (ESC to cancel)");
    if (query == NULL) return;

    int i;
    for (i = 0; i < E.numrows; i++) {
        erow *row = &E.row[i];
        char *match = strstr(row->render, query);
    }
}

```

```

    if (match) {
        E.cy = i;
        E.cx = editorRowRxToCx(row, match - row->render);
        E.rowoff = E.numrows;
        break;
    }
}

free(query);
}

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

```

[↪](#) compiles, but with no observable effects

Finally, let's map `ctrl-f` to the `editorFind()` function, and add it to the help message we set in `main()`.

kilo.c

Step 134

ctrl-f

```

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

char *editorPrompt(char *prompt) { ... }

void editorMoveCursor(int key) { ... }

void editorProcessKeypress() {
    static int quit_times = KILO_QUIT_TIMES;

    int c = editorReadKey();

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

        case CTRL_KEY('q'):
            if (E.dirty && quit_times > 0) {
                editorSetStatusMessage("WARNING!!! File has unsaved changes. "
                    "Press Ctrl-Q %d more times to quit.", quit_times);
            }

```

```

        quit_times--;
        return;
    }
    write(STDOUT_FILENO, "\x1b[2J", 4);
    write(STDOUT_FILENO, "\x1b[H", 3);
    exit(0);
    break;

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

case HOME_KEY:
    E.cx = 0;
    break;

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

case CTRL_KEY('f'):
    editorFind();
    break;

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

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

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

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

```

```

        break;

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

    default:
        editorInsertChar(c);
        break;
}

quit_times = KILO_QUIT_TIMES;
}

/** init */

void initEditor() { ... }

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

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

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

    return 0;
}

```

[↗ compiles](#)

Incremental search

Now, let's make our search feature fancy. We want to support incremental search, meaning the file is searched after each keypress when the user is typing in their search query.

To implement this, we're going to get `editorPrompt()` to take a callback function as an argument. We'll have it call this function after each keypress, passing the current search query inputted by the user and the last key they pressed.

kilo.c

Step 135

prompt-callback

```

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

```

```

/** data */
/** prototypes */

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

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

char *editorPrompt(char *prompt, void (*callback)(char *, int)) {
    size_t bufsize = 128;
    char *buf = malloc(bufsize);

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

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

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

        if (callback) callback(buf, c);
    }
}

```

```

void editorMoveCursor(int key) { ... }

void editorProcessKeypress() { ... }

/**** init ****/

```

⌘ doesn't compile

The `if` statements allow the caller to pass `NULL` for the callback, in case they don't want to use a callback. This is the case when we prompt the user for a filename, so let's pass `NULL` to `editorPrompt()` when we do that. We'll also pass `NULL` to `editorPrompt()` in `editorFind()` for now, to get the code to compile.

kilo.c

Step 136

null-callback

```

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

char *editorRowsToString(int *buflen) { ... }

void editorOpen(char *filename) { ... }

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

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

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

```

```

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

/* find */

void editorFind() {
    char *query = editorPrompt("Search: %s (ESC to cancel)", NULL);
    if (query == NULL) return;

    int i;
    for (i = 0; i < E.numrows; i++) {
        erow *row = &E.row[i];
        char *match = strstr(row->render, query);
        if (match) {
            E.cy = i;
            E.cx = editorRowRxToCx(row, match - row->render);
            E.rowoff = E.numrows;
            break;
        }
    }

    free(query);
}

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

```

[↪](#) compiles, but with no observable effects

Now let's move the actual searching code from `editorFind()` into a function called `editorFindCallback()`. Obviously this will be our callback function for `editorPrompt()`.

kilo.c

Step 137

incremental-search

```

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

void editorFindCallback(char *query, int key) {
    if (key == '\r' || key == '\x1b') {
        return;
    }
}

```

```

int i;
for (i = 0; i < E.numrows; i++) {
    erow *row = &E.row[i];
    char *match = strstr(row->render, query);
    if (match) {
        E.cy = i;
        E.cx = editorRowRxToCx(row, match - row->render);
        E.rowoff = E.numrows;
        break;
    }
}

void editorFind() {
    char *query = editorPrompt("Search: %s (ESC to cancel)", editorFindCallback);

    if (query) {
        free(query);
    }
}

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

```

[↗](#) compiles

In the callback, we check if the user pressed `Enter` or `Escape`, in which case they are leaving search mode so we return immediately instead of doing another search. Otherwise, after any other keypress, we do another search for the current query string.

That's all there is to it. We now have incremental search.

Restore cursor position when cancelling search

When the user presses `Escape` to cancel a search, we want the cursor to go back to where it was when they started the search. To do that, we'll have to save their cursor position and scroll position, and restore those values after the search is cancelled.

kilo.c

Step 138

restore-cursor

```

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

```

```

void editorFindCallback(char *query, int key) { ... }

void editorFind() {
    int saved_cx = E.cx;
    int saved_cy = E.cy;
    int saved_coloff = E.coloff;
    int saved_rowoff = E.rowoff;

    char *query = editorPrompt("Search: %s (ESC to cancel)", editorFindCallback);

    if (query) {
        free(query);
    } else {
        E.cx = saved_cx;
        E.cy = saved_cy;
        E.coloff = saved_coloff;
        E.rowoff = saved_rowoff;
    }
}

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

```

[↗ compiles](#)

If query is NULL, that means they pressed Escape, so in that case we restore the values we saved.

Search forward and backward

The last feature we'd like to add is to allow the user to advance to the next or previous match in the file using the arrow keys. The ↑ and ← keys will go to the previous match, and the ↓ and → keys will go to the next match.

We'll implement this feature using two static variables in our callback. `last_match` will contain the index of the row that the last match was on, or -1 if there was no last match. And `direction` will store the direction of the search: 1 for searching forward, and -1 for searching backward.

kilo.c

Step 139

callback-statics

```

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

```

```

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

void editorFindCallback(char *query, int key) {
    static int last_match = -1;
    static int direction = 1;

    if (key == '\r' || key == '\x1b') {
        last_match = -1;
        direction = 1;
        return;
    } else if (key == ARROW_RIGHT || key == ARROW_DOWN) {
        direction = 1;
    } else if (key == ARROW_LEFT || key == ARROW_UP) {
        direction = -1;
    } else {
        last_match = -1;
        direction = 1;
    }

    int i;
    for (i = 0; i < E.numrows; i++) {
        erow *row = &E.row[i];
        char *match = strstr(row->render, query);
        if (match) {
            E.cy = i;
            E.cx = editorRowRxToCx(row, match - row->render);
            E.rowoff = E.numrows;
            break;
        }
    }
}

void editorFind() { ... }

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

```

[↗](#) compiles, but with no observable effects

As you can see, we always reset `last_match` to `-1` unless an arrow key was pressed. So we'll only advance to the next or previous match when an arrow key is pressed. You can also see that we always set `direction` to `1` unless the `←` or `↑` key was pressed. So we always search in the forward direction unless the user specifically asks to search backwards from the last match.

If key is `'\r'` (`Enter`) or `'\x1b'` (`Escape`), that means we're about to leave search mode. So we reset `last_match` and `direction` to their initial values to get ready for the next search operation.

Now that we have those variables all set up, let's put them to use.

kilo.c

Step 140

search-arrows

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

void editorFindCallback(char *query, int key) {
    static int last_match = -1;
    static int direction = 1;

    if (key == '\r' || key == '\x1b') {
        last_match = -1;
        direction = 1;
        return;
    } else if (key == ARROW_RIGHT || key == ARROW_DOWN) {
        direction = 1;
    } else if (key == ARROW_LEFT || key == ARROW_UP) {
        direction = -1;
    } else {
        last_match = -1;
        direction = 1;
    }

    if (last_match == -1) direction = 1;
    int current = last_match;
    int i;
    for (i = 0; i < E.numrows; i++) {
        current += direction;
        if (current == -1) current = E.numrows - 1;
        else if (current == E.numrows) current = 0;

        erow *row = &E.row[current];
        char *match = strstr(row->render, query);
        if (match) {
            last_match = current;
            E.cy = current;
            E.cx = editorRowRxToCx(row, match - row->render);
            E.rowoff = E.numrows;
            break;
        }
    }
}

void editorFind() { ... }
```

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

[↗ compiles](#)

`current` is the index of the current row we are searching. If there was a last match, it starts on the line after (or before, if we're searching backwards). If there wasn't a last match, it starts at the top of the file and searches in the forward direction to find the first match.

The `if ... else if` causes `current` to go from the end of the file back to the beginning of the file, or vice versa, to allow a search to “wrap around” the end of a file and continue from the top (or bottom).

When we find a match, we set `last_match` to `current`, so that if the user presses the arrow keys, we'll start the next search from that point.

Finally, let's not forget to update the prompt text to let the user know they can use the arrow keys.

kilo.c

Step 141

search-arrows-help

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

```
void editorFindCallback(char *query, int key) { ... }
```

```
void editorFind() {
    int saved_cx = E.cx;
    int saved_cy = E.cy;
    int saved_coloff = E.coloff;
    int saved_rowoff = E.rowoff;

    char *query = editorPrompt("Search: %s (Use ESC/Arrows/Enter)",
                               editorFindCallback);

    if (query) {
        free(query);
    } else {
        E.cx = saved_cx;
        E.cy = saved_cy;
        E.coloff = saved_coloff;
        E.rowoff = saved_rowoff;
    }
}
```

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

[↗ compiles](#)

In the [next chapter](#), we'll implement syntax highlighting and filetype detection, to complete our text editor.

[1.0.0beta11](#) ([changelog](#))

[top of page](#)

Syntax highlighting

Colorful digits

Let's start by just getting some color on the screen, as simply as possible. We'll attempt to highlight numbers by coloring each digit character **red**.

kilo.c**Step 142**

syntax-digits

```
/** includes */
/** defines */
/** data */
/** prototypes */
/** terminal */
/** row operations */
/** editor operations */
/** file i/o */
/** find */
/** 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;
            char *c = &E.row[filerow].render[E.coloff];
            int j;
```

```

    for (j = 0; j < len; j++) {
        if (isdigit(c[j])) {
            abAppend(ab, "\x1b[31m", 5);
            abAppend(ab, &c[j], 1);
            abAppend(ab, "\x1b[39m", 5);
        } else {
            abAppend(ab, &c[j], 1);
        }
    }
}

abAppend(ab, "\x1b[K", 3);
abAppend(ab, "\r\n", 2);
}
}

void editorDrawStatusBar(struct abuf *ab) { ... }

void editorDrawMessageBar(struct abuf *ab) { ... }

void editorRefreshScreen() { ... }

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

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

```

✎ compiles

We can no longer just feed the substring of `render` that we want to print right into `abAppend()`. We'll have to do it character-by-character from now on. So we loop through the characters and use `isdigit()` on each one to test if it is a digit character. If it is, we precede it with the `<esc>[31m` escape sequence and follow it by the `<esc>[39m` sequence.

We previously used the `m` command ([Select Graphic Rendition](#)) to draw the status bar using inverted colors. Now we are using it to set the text color. The [VT100 User Guide](#) doesn't document color, so let's turn to the Wikipedia article on [ANSI escape codes](#). It includes a large table containing all the different argument codes you can use with the `m` command on various terminals. It also includes the ANSI color table with the 8 foreground/background colors available.

The first table says we can set the text color using codes 30 to 37, and reset it to the default color using 39. The color table says 0 is black, 1 is red, and so on, up to 7 which is white. Putting these together, we can set the text color to red using 31 as an argument to the `m` command. After printing the digit, we use 39 as an argument to `m` to set the text color back to normal.

Refactor syntax highlighting

Now we know how to color text, but we're going to have to do a lot more work to actually highlight entire strings, keywords, comments, and so on. We can't just decide what color to use based on the class of each character, like we're doing with digits currently. What we want to do is figure out the highlighting for each row of text before we display it, and then rehighlight a line whenever it gets changed. To do that, we need to store the highlighting of each line in an array. Let's add an array to the `erow` struct named `hl`, which stands for "highlight".

kilo.c

Step 143

syntax-refactoring

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

typedef struct erow {
    int size;
    int rsize;
    char *chars;
    char *render;
    unsigned char *hl;
} erow;

struct editorConfig { ... };

struct editorConfig E;

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

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

int editorRowRxToCx(erow *row, int rx) { ... }

void editorUpdateRow(erow *row) { ... }

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

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

    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.row[at].hl = NULL;
    editorUpdateRow(&E.row[at]);
}
```

```

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

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

void editorDelRow(int at) { ... }

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

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

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

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

```

[↗](#) compiles, but with no observable effects

`hl` is an array of unsigned `char` values, meaning integers in the range of 0 to 255. Each value in the array will correspond to a character in `render`, and will tell you whether that character is part of a string, or a comment, or a number, and so on. Let's create an enum containing the possible values that the `hl` array can contain.

kilo.c

Step 144

highlight-enum

```

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

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

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

enum editorKey { ... };

enum editorHighlight {
    HL_NORMAL = 0,
    HL_NUMBER
};

/** data */
/** prototypes */

```

```

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

```

[↪](#) compiles, but with no observable effects

For now, we'll focus on highlighting numbers only. So we want every character that's part of a number to have a corresponding `HL_NUMBER` value in the `hl` array, and we want every other value in `hl` to be `HL_NORMAL`.

Let's create a new `/**** syntax highlighting ****/` section, and create an `editorUpdateSyntax()` function in it. This function will go through the characters of an `erow` and highlight them by setting each value in the `hl` array.

kilo.c

Step 145

editor-update-syntax

```

/**** includes ****/
/**** defines ****/
/**** data ****/
/**** prototypes ****/
/**** terminal ****/

void die(const char *s) { ... }

void disableRawMode() { ... }

void enableRawMode() { ... }

int editorReadKey() { ... }

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

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

/**** syntax highlighting ****/

void editorUpdateSyntax(erow *row) {
    row->hl = realloc(row->hl, row->rsize);
    memset(row->hl, HL_NORMAL, row->rsize);

    int i;
    for (i = 0; i < row->rsize; i++) {
        if (isdigit(row->render[i])) {
            row->hl[i] = HL_NUMBER;
        }
    }
}

```

```

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

```

[↪](#) compiles, but with no observable effects

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

First we `realloc()` the needed memory, since this might be a new row or the row might be bigger than the last time we highlighted it. Notice that the size of the `hl` array is the same as the `render` array, so we use `rsz` as the amount of memory to allocate for `hl`.

Then we use `memset()` to set all characters to `HL_NORMAL` by default, before looping through the characters and setting the digits to `HL_NUMBER`. (Don't worry, we'll implement a better way of recognizing numbers soon enough, but right now we are focusing on refactoring.)

Now let's actually call `editorUpdateSyntax()`.

kilo.c

Step 146

call-update-syntax

```

/**** includes ****/
/**** defines ****/
/**** data ****/
/**** prototypes ****/
/**** terminal ****/
/**** syntax highlighting ****/
/**** row operations ****/

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

int editorRowRxToCx(erow *row, int rx) { ... }

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;

editorUpdateSyntax(row);
}

void editorInsertRow(int at, char *s, size_t len) { ... }

void editorFreeRow(erow *row) { ... }

void editorDelRow(int at) { ... }

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

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

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

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

```

[↗](#) compiles, but with no observable effects

`editorUpdateRow()` already has the job of updating the `render` array whenever the text of the row changes, so it makes sense that that's where we want to update the `hl` array. So after updating `render`, we call `editorUpdateSyntax()` at the end.

Next, let's make an `editorSyntaxToColor()` function that maps values in `hl` to the actual ANSI color codes we want to draw them with.

kilo.c

Step 147

map-colors

```

/** includes */
/** defines */
/** data */
/** prototypes */
/** terminal */
/** syntax highlighting */

void editorUpdateSyntax(erow *row) { ... }

int editorSyntaxToColor(int hl) {
    switch (hl) {

```

```

    case HL_NUMBER: return 31;
    default: return 37;
}
}

```

```

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

```

[↗](#) compiles, but with no observable effects

We return the ANSI code for “foreground red” for numbers, and “foreground white” for anything else that might slip through. (We’ll be handling HL_NORMAL separately, so editorSyntaxToColor() doesn’t need to handle it.)

Now let’s finally draw the highlighted text to the screen!

kilo.c

Step 148

use-hl

```

/**** includes ****/
/**** defines ****/
/**** data ****/
/**** prototypes ****/
/**** terminal ****/
/**** syntax highlighting ****/
/**** row operations ****/
/**** editor operations ****/
/**** file i/o ****/
/**** find ****/
/**** 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;
    char *c = &E.row[filerow].render[E.coloff];
    unsigned char *hl = &E.row[filerow].hl[E.coloff];
    int j;
    for (j = 0; j < len; j++) {
        if (hl[j] == HL_NORMAL) {
            abAppend(ab, "\x1b[39m", 5);
            abAppend(ab, &c[j], 1);
        } else {
            int color = editorSyntaxToColor(hl[j]);
            char buf[16];
            int clen = snprintf(buf, sizeof(buf), "\x1b[%dm", color);
            abAppend(ab, buf, clen);
            abAppend(ab, &c[j], 1);
        }
    }
    abAppend(ab, "\x1b[39m", 5);
}

abAppend(ab, "\x1b[K", 3);
abAppend(ab, "\r\n", 2);
}
}

void editorDrawStatusBar(struct abuf *ab) { ... }

void editorDrawMessageBar(struct abuf *ab) { ... }

void editorRefreshScreen() { ... }

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

/* input */
/* init */

```

[↗](#) compiles, but with no observable effects

First we get a pointer, `hl`, to the slice of the `hl` array that corresponds to the slice of `render` that we are printing. Then, for each character, if it's an `HL_NORMAL` character, we use `<esc>[39m` to make sure we're using the default text color before printing it. If it's not `HL_NORMAL`, we use `snprintf()` to write the escape sequence into a buffer which we pass to `abAppend()` before appending the actual character. Finally, after

we're done looping through all the characters and displaying them, we print a final `<esc>[39m` escape sequence to make sure the text color is reset to default.

This works, but do we really have to write out an escape sequence before every single character? In practice, most characters are going to be the same color as the previous character, so most of the escape sequences are redundant. Let's keep track of the current text color as we loop through the characters, and only print out an escape sequence when the color changes.

kilo.c

Step 149

current-color

```
/* includes */
/* defines */
/* data */
/* prototypes */
/* terminal */
/* syntax highlighting */
/* row operations */
/* editor operations */
/* file i/o */
/* find */
/* 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-- > 0) 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;
            char *c = &E.row[filerow].render[E.coloff];
            unsigned char *hl = &E.row[filerow].hl[E.coloff];
```

```

int current_color = -1;
int j;
for (j = 0; j < len; j++) {
    if (hl[j] == HL_NORMAL) {
        if (current_color != -1) {
            abAppend(ab, "\x1b[39m", 5);
            current_color = -1;
        }
        abAppend(ab, &c[j], 1);
    } else {
        int color = editorSyntaxToColor(hl[j]);
        if (color != current_color) {
            current_color = color;
            char buf[16];
            int clen = snprintf(buf, sizeof(buf), "\x1b[%dm", color);
            abAppend(ab, buf, clen);
        }
        abAppend(ab, &c[j], 1);
    }
}
abAppend(ab, "\x1b[39m", 5);
}

abAppend(ab, "\x1b[K", 3);
abAppend(ab, "\r\n", 2);
}
}

void editorDrawStatusBar(struct abuf *ab) { ... }

void editorDrawMessageBar(struct abuf *ab) { ... }

void editorRefreshScreen() { ... }

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

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

```

[↗](#) compiles, but with no observable effects

`current_color` is `-1` when we want the default text color, otherwise it is set to the value that `editorSyntaxToColor()` last returned. When the color changes, we print out the escape sequence for that color and set `current_color` to the new color. When we go from highlighted text back to `HL_NORMAL` text, we print out the `<esc>[39m` escape sequence and set `current_color` to `-1`.

That concludes our refactoring of the syntax highlighting system.

Colorful search results

Before we start highlighting strings and keywords and all that, let's use our highlighting system to highlight search results. We'll start by adding `HL_MATCH` to the `editorHighlight` enum, and mapping it to the color blue (34) in `editorSyntaxToColor()`.

kilo.c

Step 150

hl-match

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

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

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

enum editorKey { ... };

enum editorHighlight {
    HL_NORMAL = 0,
    HL_NUMBER,
    HL_MATCH
};

/**/ data ***/
/**/ prototypes ***/
/**/ terminal ***/
/**/ syntax highlighting ***/

void editorUpdateSyntax(erow *row) { ... }

int editorSyntaxToColor(int hl) {
    switch (hl) {
        case HL_NUMBER: return 31;
        case HL_MATCH: return 34;
        default: return 37;
    }
}

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

[↗](#) compiles, but with no observable effects

Now all we have to do is `memset()` the matched substring to `HL_MATCH` in our search code.

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

void editorFindCallback(char *query, int key) {
    static int last_match = -1;
    static int direction = 1;

    if (key == '\r' || key == '\x1b') {
        last_match = -1;
        direction = 1;
        return;
    } else if (key == ARROW_RIGHT || key == ARROW_DOWN) {
        direction = 1;
    } else if (key == ARROW_LEFT || key == ARROW_UP) {
        direction = -1;
    } else {
        last_match = -1;
        direction = 1;
    }

    if (last_match == -1) direction = 1;
    int current = last_match;
    int i;
    for (i = 0; i < E.numrows; i++) {
        current += direction;
        if (current == -1) current = E.numrows - 1;
        else if (current == E.numrows) current = 0;

        erow *row = &E.row[current];
        char *match = strstr(row->render, query);
        if (match) {
            last_match = current;
            E.cy = current;
            E.cx = editorRowRxToCx(row, match - row->render);
            E.rowoff = E.numrows;

            memset(&row->hl[match - row->render], HL_MATCH, strlen(query));
            break;
        }
    }
}

void editorFind() { ... }
```

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

[↗](#) compiles

`match - row->render` is the index into `render` of the match, so we use that as our index into `hl`.

Restore syntax highlighting after search

Currently, search results stay highlighted in blue even after the user is done using the search feature. We want to restore `hl` to its previous value after each search. To do that, we'll save the original contents of `hl` in a static variable named `saved_hl` in `editorFindCallback()`, and restore `hl` to the contents of `saved_hl` at the top of the callback.

kilo.c

Step 152

restore-hl

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

```
void editorFindCallback(char *query, int key) {
    static int last_match = -1;
    static int direction = 1;

    static int saved_hl_line;
    static char *saved_hl = NULL;

    if (saved_hl) {
        memcpy(E.row[saved_hl_line].hl, saved_hl, E.row[saved_hl_line].rsize);
        free(saved_hl);
        saved_hl = NULL;
    }

    if (key == '\r' || key == '\x1b') {
        last_match = -1;
        direction = 1;
        return;
    } else if (key == ARROW_RIGHT || key == ARROW_DOWN) {
        direction = 1;
    } else if (key == ARROW_LEFT || key == ARROW_UP) {
        direction = -1;
    }
}
```

```

} else {
    last_match = -1;
    direction = 1;
}

if (last_match == -1) direction = 1;
int current = last_match;
int i;
for (i = 0; i < E.numrows; i++) {
    current += direction;
    if (current == -1) current = E.numrows - 1;
    else if (current == E.numrows) current = 0;

    erow *row = &E.row[current];
    char *match = strstr(row->render, query);
    if (match) {
        last_match = current;
        E.cy = current;
        E.cx = editorRowRxToCx(row, match - row->render);
        E.rowoff = E.numrows;

        saved_hl_line = current;
        saved_hl = malloc(row->rsz);
        memcpy(saved_hl, row->hl, row->rsz);
        memset(&row->hl[match - row->render], HL_MATCH, strlen(query));
        break;
    }
}
}

void editorFind() { ... }

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

```

[↗ compiles](#)

We use another static variable named `saved_hl_line` to know which line's `hl` needs to be restored. `saved_hl` is a dynamically allocated array which points to `NULL` when there is nothing to restore. If there is something to restore, we `memcpy()` it to the saved line's `hl` and then deallocate `saved_hl` and set it back to `NULL`.

Notice that the `malloc()`'d memory is guaranteed to be `free()`'d, because when the user closes the search prompt by pressing `Enter` or `Escape`, `editorPrompt()` calls our callback, giving a chance for `hl` to be restored before `editorPrompt()` finally returns. Also notice that it's impossible for `saved_hl` to get `malloc()`'d before its old value gets `free()`'d, because we always `free()` it at the top of the function. And finally, it's impossible for the user to edit the file between saving and

restoring the `hl`, so we can safely use `saved_hl_line` as an index into `E.row`. (It's important to think about these things.)

Colorful numbers

Alright, let's start working on highlighting numbers properly. First, we'll change our `for` loop in `editorUpdateSyntax()` to a `while` loop, to allow us to consume multiple characters each iteration. (We'll only consume one character at a time for numbers, but this will be useful for later.)

kilo.c

Step 153

syntax-while

```
/**/ includes ***/
/**/ defines ***/
/**/ data ***/
/**/ prototypes ***/
/**/ terminal ***/
/**/ syntax highlighting ***/

void editorUpdateSyntax(erow *row) {
    row->hl = realloc(row->hl, row->rsize);
    memset(row->hl, HL_NORMAL, row->rsize);

    int i = 0;
    while (i < row->rsize) {
        char c = row->render[i];

        if (isdigit(c)) {
            row->hl[i] = HL_NUMBER;
        }

        i++;
    }
}

int editorSyntaxToColor(int hl) { ... }

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

[↪](#) compiles, but with no observable effects

Now let's define an `is_separator()` function that takes a character and returns true if it's considered a separator character.

```

/** includes */
/** defines */
/** data */
/** prototypes */
/** terminal */
/** syntax highlighting */

int is_separator(int c) {
    return isspace(c) || c == '\0' || strchr(",.()+-/*=~%<>[];", c) != NULL;
}

void editorUpdateSyntax(erow *row) { ... }

int editorSyntaxToColor(int hl) { ... }

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

```

[↗](#) compiles, but with no observable effects

`strchr()` comes from `<string.h>`. It looks for the first occurrence of a character in a string, and returns a pointer to the matching character in the string. If the string doesn't contain the character, `strchr()` returns `NULL`.

Right now, numbers are highlighted even if they're part of an identifier, such as the 32 in `int32_t`. To fix that, we'll require that numbers are preceded by a separator character, which includes whitespace or punctuation characters. We also include the null byte (`'\0'`), because then we can count the null byte at the end of each line as a separator, which will make some of our code simpler in the future.

Let's add a `prev_sep` variable to `editorUpdateSyntax()` that keeps track of whether the previous character was a separator. Then let's use it to recognize and highlight numbers properly.

```

/** includes */
/** defines */
/** data */
/** prototypes */
/** terminal */
/** syntax highlighting */

int is_separator(int c) { ... }

```

```

void editorUpdateSyntax(erow *row) {
    row->hl = realloc(row->hl, row->rsize);
    memset(row->hl, HL_NORMAL, row->rsize);

    int prev_sep = 1;

    int i = 0;
    while (i < row->rsize) {
        char c = row->render[i];
        unsigned char prev_hl = (i > 0) ? row->hl[i - 1] : HL_NORMAL;

        if (isdigit(c) && (prev_sep || prev_hl == HL_NUMBER)) {
            row->hl[i] = HL_NUMBER;
            i++;
            prev_sep = 0;
            continue;
        }

        prev_sep = is_separator(c);
        i++;
    }
}

int editorSyntaxToColor(int hl) { ... }

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

```

 compiles

We initialize `prev_sep` to 1 (meaning true) because we consider the beginning of the line to be a separator. (Otherwise numbers at the very beginning of the line wouldn't be highlighted.)

`prev_hl` is set to the highlight type of the previous character. To highlight a digit with `HL_NUMBER`, we now require the previous character to either be a separator, or to also be highlighted with `HL_NUMBER`.

When we decide to highlight the current character a certain way (`HL_NUMBER` in this case), we increment `i` to “consume” that character, set `prev_sep` to 0 to indicate we are in the middle of highlighting something, and then `continue` the loop. We will use this pattern for each thing that we highlight.

If we end up not highlighting the current character, then we'll end up at the bottom of the `while` loop, where we set `prev_sep` according to whether the current character

is a separator, and we increment `i` to consume the character. The `memset()` we did at the top of the function means that an unhighlighted character will have a value of `HL_NORMAL` in `hl`.

Now let's support highlighting numbers that contain decimal points.

kilo.c

Step 156

decimal-point

```
/** includes */
/** defines */
/** data */
/** prototypes */
/** terminal */
/** syntax highlighting */

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) {
    row->hl = realloc(row->hl, row->rsize);
    memset(row->hl, HL_NORMAL, row->rsize);

    int prev_sep = 1;

    int i = 0;
    while (i < row->rsize) {
        char c = row->render[i];
        unsigned char prev_hl = (i > 0) ? row->hl[i - 1] : HL_NORMAL;

        if ((isdigit(c) && (prev_sep || prev_hl == HL_NUMBER)) ||
            (c == '.' && prev_hl == HL_NUMBER)) {
            row->hl[i] = HL_NUMBER;
            i++;
            prev_sep = 0;
            continue;
        }

        prev_sep = is_separator(c);
        i++;
    }
}

int editorSyntaxToColor(int hl) { ... }
```

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

[↗](#) compiles

A . character that comes after a character that we just highlighted as a number will now be considered part of the number.

Detect filetype

Before we go on to highlight other things, we're going to add filetype detection to our editor. This will allow us to have different rules for how to highlight different types of files. For example, text files shouldn't have any highlighting, and C files should highlight numbers, strings, C/C++-style comments, and many different keywords specific to C.

Let's create an `editorSyntax` struct that will contain all the syntax highlighting information for a particular filetype.

kilo.c

Step 157

editor-syntax

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

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

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

enum editorKey { ... };

enum editorHighlight { ... };

#define HL_HIGHLIGHT_NUMBERS (1<<0)

/** data */

struct editorSyntax {
    char *filetype;
    char **filematch;
    int flags;
};

typedef struct erow { ... } erow;

struct editorConfig { ... };

struct editorConfig E;

/** prototypes */
/** terminal */
/** syntax highlighting */
/** row operations */
/** editor operations */
```

```

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

```

[↪](#) compiles, but with no observable effects

The `filetype` field is the name of the filetype that will be displayed to the user in the status bar. `filematch` is an array of strings, where each string contains a pattern to match a filename against. If the filename matches, then the file will be recognized as having that filetype. Finally, `flags` is a bit field that will contain flags for whether to highlight numbers and whether to highlight strings for that filetype. For now, we define just the `HL_HIGHLIGHT_NUMBERS` flag bit.

Now let's make an array of built-in `editorSyntax` structs, and add one for the C language to it.

kilo.c

Step 158

hlDb

```

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

struct editorSyntax { ... };

typedef struct erow { ... } erow;

struct editorConfig { ... };

struct editorConfig E;

/**** filetypes ****/

char *C_HL_extensions[] = { ".c", ".h", ".cpp", NULL };

struct editorSyntax HLDB[] = {
    {
        "c",
        C_HL_extensions,
        HL_HIGHLIGHT_NUMBERS
    },
};

#define HLDB_ENTRIES (sizeof(HLDB) / sizeof(HLDB[0]))

/**** prototypes ****/
/**** terminal ****/
/**** syntax highlighting ****/
/**** row operations ****/
/**** editor operations ****/

```

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

[↗](#) compiles, but with no observable effects

HLDB stands for “highlight database”. Our `editorSyntax` struct for the C language contains the string “c” for the `filetype` field, the extensions “.c”, “.h”, and “.cpp” for the `filematch` field (the array must be terminated with `NULL`), and the `HL_HIGHLIGHT_NUMBERS` flag turned on in the `flags` field.

We then define an `HLDB_ENTRIES` constant to store the length of the `HLDB` array.

Now let’s add a pointer to the current `editorSyntax` struct in our global editor state, and initialize it to `NULL`.

kilo.c

Step 159

e-syntax

```
/**/ includes ***/  
/**/ defines ***/  
/**/ data ***/  
  
struct editorSyntax { ... };  
  
typedef struct erow { ... } erow;  
  
struct editorConfig {  
    int cx, cy;  
    int rx;  
    int rowoff;  
    int coloff;  
    int screenrows;  
    int screencols;  
    int numrows;  
    erow *row;  
    int dirty;  
    char *filename;  
    char statusmsg[80];  
    time_t statusmsg_time;  
    struct editorSyntax *syntax;  
    struct termios orig_termios;  
};  
  
struct editorConfig E;  
  
/**/ filetypes ***/  
/**/ prototypes ***/  
/**/ terminal ***/  
/**/ syntax highlighting ***/  
/**/ row operations ***/
```

```

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

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

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

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

```

[↗](#) compiles, but with no observable effects

When `E.syntax` is `NULL`, that means there is no filetype for the current file, and no syntax highlighting should be done.

Let's show the current filetype in the status bar. If `E.syntax` is `NULL`, then we'll display `no ft` ("no filetype") instead.

kilo.c

Step 160

show-filetype

```

/**** includes ****/
/**** defines ****/
/**** data ****/
/**** filetypes ****/
/**** prototypes ****/
/**** terminal ****/
/**** syntax highlighting ****/
/**** row operations ****/
/**** editor operations ****/
/**** file i/o ****/
/**** find ****/
/**** append buffer ****/
/**** output ****/

void editorScroll() { ... }

```

```

void editorDrawRows(struct abuf *ab) { ... }

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

void editorDrawMessageBar(struct abuf *ab) { ... }

void editorRefreshScreen() { ... }

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

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

```

[↗ compiles](#)

Now let's change `editorUpdateSyntax()` to take the current `E.syntax` value into account.

kilo.c

Step 161

use-filetype

```

/** includes */
/** defines */
/** data */
/** filetypes */
/** prototypes */
/** terminal */
/** syntax highlighting */

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) {
    row->hl = realloc(row->hl, row->rsiz);
    memset(row->hl, HL_NORMAL, row->rsiz);
}

```

```

if (E.syntax == NULL) return;

int prev_sep = 1;

int i = 0;
while (i < row->rsz) {
    char c = row->render[i];
    unsigned char prev_hl = (i > 0) ? row->hl[i - 1] : HL_NORMAL;

    if (E.syntax->flags & HL_HIGHLIGHT_NUMBERS) {
        if ((isdigit(c) && (prev_sep || prev_hl == HL_NUMBER)) ||
            (c == '.' && prev_hl == HL_NUMBER)) {
            row->hl[i] = HL_NUMBER;
            i++;
            prev_sep = 0;
            continue;
        }
    }

    prev_sep = is_separator(c);
    i++;
}

int editorSyntaxToColor(int hl) { ... }

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

```

[↗ compiles](#)

If no filetype is set, we return immediately after `memset()`ing the entire line to `HL_NORMAL`. We also wrap the number-highlighting code in an `if` statement that checks to see if numbers should be highlighted for the current filetype.

Now we'll create an `editorSelectSyntaxHighlight()` function that tries to match the current filename to one of the `filematch` fields in the `HLDB`. If one matches, it'll set `E.syntax` to that filetype.

kilo.c

Step 162

select-syntax

```

/* includes */
/* defines */
/* data */
/* filetypes */
/* prototypes */

```

```

/** terminal */
/** syntax highlighting */

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) { ... }

int editorSyntaxToColor(int hl) { ... }

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

    char *ext = strrchr(E.filename, '.');

    for (unsigned int j = 0; j < HLDB_ENTRIES; j++) {
        struct editorSyntax *s = &HLDB[j];
        unsigned int i = 0;
        while (s->filematch[i]) {
            int is_ext = (s->filematch[i][0] == '.');
            if ((is_ext && ext && !strcmp(ext, s->filematch[i])) ||
                (!is_ext && strstr(E.filename, s->filematch[i]))) {
                E.syntax = s;
                return;
            }
            i++;
        }
    }
}

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

```

[↗](#) compiles, but with no observable effects

`strrchr()` and `strcmp()` come from `<string.h>`. `strrchr()` returns a pointer to the last occurrence of a character in a string, and `strcmp()` returns 0 if two given strings are equal.

First we set `E.syntax` to `NULL`, so that if nothing matches or if there is no filename, then there is no filetype.

Then we get a pointer to the extension part of the filename by using `strrchr()` to find the last occurrence of the `.` character. If there is no extension, then `ext` will be `NULL`.

Finally, we loop through each `editorSyntax` struct in the `HLDB` array, and for each one of those, we loop through each pattern in its `filematch` array. If the pattern starts with a `.`, then it's a file extension pattern, and we use `strcmp()` to see if the filename ends with that extension. If it's not a file extension pattern, then we just check to see if the pattern exists anywhere in the filename, using `strstr()`. If the filename matched according to those rules, then we set `E.syntax` to the current `editorSyntax` struct, and return.

We want to call `editorSelectSyntaxHighlight()` wherever `E.filename` changes. This is in `editorOpen()` and `editorSave()`.

kilo.c

Step 163

detect-filetype

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

char *editorRowsToString(int *buflen) { ... }

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

    editorSelectSyntaxHighlight();

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

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

void editorSave() {
    if (E.filename == NULL) {
```

```

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

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

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

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

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

```

[↗ compiles](#)

At this point, when you open a C file in the editor, you should see numbers getting highlighted, and you should see `c` in the status bar where we display the filetype. When you start up the editor with no arguments and save the file with a filename that ends in `.c`, you should see the filetype in the status bar change satisfyingly from `no ft` to `c`. However, any numbers you might have in the file will not be highlighted! Very unsatisfying!

Let's rehighlight the entire file after setting `E.syntax` in `editorSelectSyntaxHighlight()`.

kilo.c

Step 164

rehighlight

```

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

```

```

/** filetypes */
/** prototypes */
/** terminal */
/** syntax highlighting */

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) { ... }

int editorSyntaxToColor(int hl) { ... }

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

    char *ext = strrchr(E.filename, '.');

    for (unsigned int j = 0; j < HLDB_ENTRIES; j++) {
        struct editorSyntax *s = &HLDB[j];
        unsigned int i = 0;
        while (s->filematch[i]) {
            int is_ext = (s->filematch[i][0] == '.');
            if ((is_ext && ext && !strcmp(ext, s->filematch[i])) ||
                (!is_ext && strstr(E.filename, s->filematch[i]))) {
                E.syntax = s;

                int filerow;
                for (filerow = 0; filerow < E.numrows; filerow++) {
                    editorUpdateSyntax(&E.row[filerow]);
                }

                return;
            }
            i++;
        }
    }
}

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

```

 compiles

We simply loop through each row in the file, and call `editorUpdateSyntax()` on it. Now the highlighting immediately changes when the filetype changes.

Colorful strings

With all that out of the way, we can finally get to highlighting more things! Let's start with strings.

kilo.c

Step 165

hl-string

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

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

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

enum editorKey { ... };

enum editorHighlight {
    HL_NORMAL = 0,
    HL_STRING,
    HL_NUMBER,
    HL_MATCH
};

#define HL_HIGHLIGHT_NUMBERS (1<<0)

*** data ***
*** filetypes ***
*** prototypes ***
*** terminal ***
*** syntax highlighting ***

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) { ... }

int editorSyntaxToColor(int hl) {
    switch (hl) {
        case HL_STRING: return 35;
        case HL_NUMBER: return 31;
        case HL_MATCH: return 34;
        default: return 37;
    }
}

void editorSelectSyntaxHighlight() { ... }

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

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

[↗](#) compiles, but with no observable effects

We're coloring strings magenta (35).

Now let's add an HL_HIGHLIGHT_STRINGS bit flag to the flags field of the editorSyntax struct, and turn on the flag when highlighting C files.

kilo.c

Step 166

string-flag

```
/**/ includes ***/  
/**/ defines ***/  
  
#define KILO_VERSION "0.0.1"  
#define KILO_TAB_STOP 8  
#define KILO_QUIT_TIMES 3  
  
#define CTRL_KEY(k) ((k) & 0x1f)  
  
enum editorKey { ... };  
  
enum editorHighlight { ... };  
  
#define HL_HIGHLIGHT_NUMBERS (1<<0)  
#define HL_HIGHLIGHT_STRINGS (1<<1)  
  
/**/ data ***/  
/**/ filetypes ***/  
  
char *C_HL_extensions[] = { ".c", ".h", ".cpp", NULL };  
  
struct editorSyntax HLDB[] = {  
    {  
        "c",  
        C_HL_extensions,  
        HL_HIGHLIGHT_NUMBERS | HL_HIGHLIGHT_STRINGS  
    },  
};  
  
#define HLDB_ENTRIES (sizeof(HLDB) / sizeof(HLDB[0]))  
  
/**/ prototypes ***/  
/**/ terminal ***/  
/**/ syntax highlighting ***/  
/**/ row operations ***/  
/**/ editor operations ***/  
/**/ file i/o ***/  
/**/ find ***/  
/**/ append buffer ***/  
/**/ output ***/  
/**/ input ***/
```

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

[↗](#) compiles, but with no observable effects

Now for the actual highlighting code. We will use an `in_string` variable to keep track of whether we are currently inside a string. If we are, then we'll keep highlighting the current character as a string until we hit the closing quote.

kilo.c

Step 167

syntax-strings

```
/**/ includes */
/**/ defines */
/**/ data */
/**/ filetypes */
/**/ prototypes */
/**/ terminal */
/**/ syntax highlighting */

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) {
    row->hl = realloc(row->hl, row->rsize);
    memset(row->hl, HL_NORMAL, row->rsize);

    if (E.syntax == NULL) return;

    int prev_sep = 1;
    int in_string = 0;

    int i = 0;
    while (i < row->rsize) {
        char c = row->render[i];
        unsigned char prev_hl = (i > 0) ? row->hl[i - 1] : HL_NORMAL;

        if (E.syntax->flags & HL_HIGHLIGHT_STRINGS) {
            if (in_string) {
                row->hl[i] = HL_STRING;
                if (c == in_string) in_string = 0;
                i++;
                prev_sep = 1;
                continue;
            } else {
                if (c == '"' || c == '\\') {
                    in_string = c;
                    row->hl[i] = HL_STRING;
                    i++;
                    continue;
                }
            }
        }

        if (E.syntax->flags & HL_HIGHLIGHT_NUMBERS) {
            if ((isdigit(c) && (prev_sep || prev_hl == HL_NUMBER)) ||
                (c == '.' && prev_hl == HL_NUMBER)) {
```

```

        row->hl[i] = HL_NUMBER;
        i++;
        prev_sep = 0;
        continue;
    }
}

prev_sep = is_separator(c);
i++;
}
}

int editorSyntaxToColor(int hl) { ... }

void editorSelectSyntaxHighlight() { ... }

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

```

[↗ compiles](#)

As you can see, we highlight both double-quoted strings and single-quoted strings (sorry Lispers/Rustaceans). We actually store either a double-quote (") or a single-quote (') character as the value of `in_string`, so that we know which one closes the string.

So, going through the code from top to bottom: If `in_string` is set, then we know the current character can be highlighted with `HL_STRING`. Then we check if the current character is the closing quote (`c == in_string`), and if so, we reset `in_string` to 0. Then, since we highlighted the current character, we have to consume it by incrementing `i` and continuing out of the current loop iteration. We also set `prev_sep` to 1 so that if we're done highlighting the string, the closing quote is considered a separator.

If we're not currently in a string, then we have to check if we're at the beginning of one by checking for a double- or single-quote. If we are, we store the quote in `in_string`, highlight it with `HL_STRING`, and consume it.

We should probably take escaped quotes into account when highlighting strings. If the sequence `\'` or `\"` occurs in a string, then the escaped quote doesn't close the string in the vast majority of languages.

```
/* includes */
/* defines */
/* data */
/* filetypes */
/* prototypes */
/* terminal */
/* syntax highlighting */
```

```
int is_separator(int c) { ... }
```

```
void editorUpdateSyntax(erow *row) {
    row->hl = realloc(row->hl, row->rsize);
    memset(row->hl, HL_NORMAL, row->rsize);

    if (E.syntax == NULL) return;

    int prev_sep = 1;
    int in_string = 0;

    int i = 0;
    while (i < row->rsize) {
        char c = row->render[i];
        unsigned char prev_hl = (i > 0) ? row->hl[i - 1] : HL_NORMAL;

        if (E.syntax->flags & HL_HIGHLIGHT_STRINGS) {
            if (in_string) {
                row->hl[i] = HL_STRING;
                if (c == '\\') && i + 1 < row->rsize) {
                    row->hl[i + 1] = HL_STRING;
                    i += 2;
                    continue;
                }
            }
            if (c == in_string) in_string = 0;
            i++;
            prev_sep = 1;
            continue;
        } else {
            if (c == '"' || c == '\\') {
                in_string = c;
                row->hl[i] = HL_STRING;
                i++;
                continue;
            }
        }
    }

    if (E.syntax->flags & HL_HIGHLIGHT_NUMBERS) {
        if ((isdigit(c) && (prev_sep || prev_hl == HL_NUMBER)) ||
            (c == '.' && prev_hl == HL_NUMBER)) {
            row->hl[i] = HL_NUMBER;
            i++;
            prev_sep = 0;
            continue;
        }
    }
}
```

```

    }
}

prev_sep = is_separator(c);
i++;
}
}

int editorSyntaxToColor(int hl) { ... }

void editorSelectSyntaxHighlight() { ... }

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

```

[↗ compiles](#)

If we're in a string and the current character is a backslash (\), *and* there's at least one more character in that line that comes after the backslash, then we highlight the character that comes after the backslash with HL_STRING and consume it. We increment i by 2 to consume both characters at once.

Colorful single-line comments

Next let's highlight single-line comments. (We'll leave multi-line comments until the end, because they're complicated.)

kilo.c

Step 169

hl-comment

```

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

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

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

enum editorKey { ... };

enum editorHighlight {
    HL_NORMAL = 0,
    HL_COMMENT,
    HL_STRING,
    HL_NUMBER,
    HL_MATCH

```

```

};

#define HL_HIGHLIGHT_NUMBERS (1<<0)
#define HL_HIGHLIGHT_STRINGS (1<<1)

/**/ data ***/
/**/ filetypes ***/
/**/ prototypes ***/
/**/ terminal ***/
/**/ syntax highlighting ***/

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) { ... }

int editorSyntaxToColor(int hl) {
    switch (hl) {
        case HL_COMMENT: return 36;
        case HL_STRING: return 35;
        case HL_NUMBER: return 31;
        case HL_MATCH: return 34;
        default: return 37;
    }
}

void editorSelectSyntaxHighlight() { ... }

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

```

[↪](#) compiles, but with no observable effects

Comments will be highlighted in cyan (36).

We'll let each language specify its own single-line comment pattern, as they differ a lot between languages. Let's add a `singleline_comment_start` string to the `editorSyntax` struct, and set it to `"//"` for the C filetype.

kilo.c

Step 170

scs

```

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

struct editorSyntax {
    char *filetype;
    char **filematch;

```

```

char *singleline_comment_start;
int flags;
};

typedef struct erow { ... } erow;

struct editorConfig { ... };

struct editorConfig E;

/** filetypes */

char *C_HL_extensions[] = { ".c", ".h", ".cpp", NULL };

struct editorSyntax HLDB[] = {
    {
        "c",
        C_HL_extensions,
        "//",
        HL_HIGHLIGHT_NUMBERS | HL_HIGHLIGHT_STRINGS
    },
};

#define HLDB_ENTRIES (sizeof(HLDB) / sizeof(HLDB[0]))

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

```

[↪](#) compiles, but with no observable effects

Okay, now for the highlighting code.

kilo.c

Step 171

syntax-comments

```

/** includes */
/** defines */
/** data */
/** filetypes */
/** prototypes */
/** terminal */
/** syntax highlighting */

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) {

```

```

row->hl = realloc(row->hl, row->rsize);
memset(row->hl, HL_NORMAL, row->rsize);

if (E.syntax == NULL) return;

char *scs = E.syntax->singleline_comment_start;
int scs_len = scs ? strlen(scs) : 0;

int prev_sep = 1;
int in_string = 0;

int i = 0;
while (i < row->rsize) {
    char c = row->render[i];
    unsigned char prev_hl = (i > 0) ? row->hl[i - 1] : HL_NORMAL;

    if (scs_len && !in_string) {
        if (!strncmp(&row->render[i], scs, scs_len)) {
            memset(&row->hl[i], HL_COMMENT, row->rsize - i);
            break;
        }
    }

    if (E.syntax->flags & HL_HIGHLIGHT_STRINGS) {
        if (in_string) {
            row->hl[i] = HL_STRING;
            if (c == '\\\\' && i + 1 < row->rsize) {
                row->hl[i + 1] = HL_STRING;
                i += 2;
                continue;
            }
            if (c == in_string) in_string = 0;
            i++;
            prev_sep = 1;
            continue;
        } else {
            if (c == '"' || c == '\\') {
                in_string = c;
                row->hl[i] = HL_STRING;
                i++;
                continue;
            }
        }
    }

    if (E.syntax->flags & HL_HIGHLIGHT_NUMBERS) {
        if ((isdigit(c) && (prev_sep || prev_hl == HL_NUMBER)) ||
            (c == '.' && prev_hl == HL_NUMBER)) {
            row->hl[i] = HL_NUMBER;
            i++;
            prev_sep = 0;
            continue;
        }
    }
}

```

```

    }

    prev_sep = is_separator(c);
    i++;
}
}

int editorSyntaxToColor(int hl) { ... }

void editorSelectSyntaxHighlight() { ... }

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

```

[↗ compiles](#)

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

If you don't want single-line comment highlighting for a particular filetype, you should be able to set `singleline_comment_start` either to `NULL` or to the empty string `("")`. We make `scs` an alias for `E.syntax->singleline_comment_start` for easier typing (and readability, perhaps?). We then set `scs_len` to the length of the string, or `0` if the string is `NULL`. This lets us use `scs_len` as a boolean to know whether we should highlight single-line comments.

So we wrap our comment highlighting code in an `if` statement that checks `scs_len` and also makes sure we're not in a string, since we're placing this code above the string highlighting code (order matters a lot in this function).

If those checks passed, then we use `strncmp()` to check if this character is the start of a single-line comment. If so, then we simply `memset()` the whole rest of the line with `HL_COMMENT` and `break` out of the syntax highlighting loop. Just like that, we're done highlighting the line.

Colorful keywords

Now let's turn to highlighting keywords. We're going to allow languages to specify two types of keywords that will be highlighted in different colors. (In C, we'll highlight actual keywords in one color and common type names in the other color.)

kilo.c

Step 172

hl-keywords

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

```

/** defines */

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

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

enum editorKey { ... };

enum editorHighlight {
    HL_NORMAL = 0,
    HL_COMMENT,
    HL_KEYWORD1,
    HL_KEYWORD2,
    HL_STRING,
    HL_NUMBER,
    HL_MATCH
};

#define HL_HIGHLIGHT_NUMBERS (1<<0)
#define HL_HIGHLIGHT_STRINGS (1<<1)

/** data */
/** filetypes */
/** prototypes */
/** terminal */
/** syntax highlighting */

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) { ... }

int editorSyntaxToColor(int hl) {
    switch (hl) {
        case HL_COMMENT: return 36;
        case HL_KEYWORD1: return 33;
        case HL_KEYWORD2: return 32;
        case HL_STRING: return 35;
        case HL_NUMBER: return 31;
        case HL_MATCH: return 34;
        default: return 37;
    }
}

void editorSelectSyntaxHighlight() { ... }

/** row operations */
/** editor operations */
/** file i/o */
/** find */
/** append buffer */
/** output */

```

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

[↗](#) compiles, but with no observable effects

The two colors we'll use for keywords are yellow (33) and green (32).

Let's add a keywords array to the editorSyntax struct. This will be a NULL - terminated array of strings, each string containing a keyword. To differentiate between the two types of keywords, we'll terminate the second type of keywords with a pipe (|) character (also known as a vertical bar).

kilo.c

Step 173

c-keywords

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

struct editorSyntax {
    char *filetype;
    char **filematch;
    char **keywords;
    char *singleline_comment_start;
    int flags;
};

typedef struct erow { ... } erow;

struct editorConfig { ... };

struct editorConfig E;

/**/ filetypes */*/

char *C_HL_extensions[] = { ".c", ".h", ".cpp", NULL };
char *C_HL_keywords[] = {
    "switch", "if", "while", "for", "break", "continue", "return", "else",
    "struct", "union", "typedef", "static", "enum", "class", "case",

    "int|", "long|", "double|", "float|", "char|", "unsigned|", "signed|",
    "void|", NULL
};

struct editorSyntax HLDB[] = {
    {
        "c",
        C_HL_extensions,
        C_HL_keywords,
        "//",
        HL_HIGHLIGHT_NUMBERS | HL_HIGHLIGHT_STRINGS
    },
};
```

```
#define HLDB_ENTRIES (sizeof(HLDB) / sizeof(HLDB[0]))
```

```
/**** prototypes ****/  
/**** terminal ****/  
/**** syntax highlighting ****/  
/**** row operations ****/  
/**** editor operations ****/  
/**** file i/o ****/  
/**** find ****/  
/**** append buffer ****/  
/**** output ****/  
/**** input ****/  
/**** init ****/
```

[↪](#) compiles, but with no observable effects

As mentioned earlier, we'll highlight common C types as secondary keywords, so we end each one with a `|` character.

Now let's highlight them.

kilo.c

Step 174

syntax-keywords

```
/**** includes ****/  
/**** defines ****/  
/**** data ****/  
/**** filetypes ****/  
/**** prototypes ****/  
/**** terminal ****/  
/**** syntax highlighting ****/  
  
int is_separator(int c) { ... }  
  
void editorUpdateSyntax(erow *row) {  
    row->hl = realloc(row->hl, row->rsize);  
    memset(row->hl, HL_NORMAL, row->rsize);  
  
    if (E.syntax == NULL) return;  
  
    char **keywords = E.syntax->keywords;  
  
    char *scs = E.syntax->singleline_comment_start;  
    int scs_len = scs ? strlen(scs) : 0;  
  
    int prev_sep = 1;  
    int in_string = 0;  
  
    int i = 0;  
    while (i < row->rsize) {  
        char c = row->render[i];  
        unsigned char prev_hl = (i > 0) ? row->hl[i - 1] : HL_NORMAL;  
  
        if (scs_len && !in_string) {  
            if (!strncmp(&row->render[i], scs, scs_len)) {
```

```

        memset(&row->hl[i], HL_COMMENT, row->rsize - i);
        break;
    }
}

if (E.syntax->flags & HL_HIGHLIGHT_STRINGS) {
    if (in_string) {
        row->hl[i] = HL_STRING;
        if (c == '\\') && i + 1 < row->rsize) {
            row->hl[i + 1] = HL_STRING;
            i += 2;
            continue;
        }
        if (c == in_string) in_string = 0;
        i++;
        prev_sep = 1;
        continue;
    } else {
        if (c == '"' || c == '\\') {
            in_string = c;
            row->hl[i] = HL_STRING;
            i++;
            continue;
        }
    }
}

if (E.syntax->flags & HL_HIGHLIGHT_NUMBERS) {
    if ((isdigit(c) && (prev_sep || prev_hl == HL_NUMBER)) ||
        (c == '.' && prev_hl == HL_NUMBER)) {
        row->hl[i] = HL_NUMBER;
        i++;
        prev_sep = 0;
        continue;
    }
}

if (prev_sep) {
    int j;
    for (j = 0; keywords[j]; j++) {
        int klen = strlen(keywords[j]);
        int kw2 = keywords[j][klen - 1] == '|';
        if (kw2) klen--;

        if (!strncmp(&row->render[i], keywords[j], klen) &&
            is_separator(row->render[i + klen])) {
            memset(&row->hl[i], kw2 ? HL_KEYWORD2 : HL_KEYWORD1, klen);
            i += klen;
            break;
        }
    }
    if (keywords[j] != NULL) {
        prev_sep = 0;
    }
}

```

```

        continue;
    }
}

prev_sep = is_separator(c);
i++;
}
}

int editorSyntaxToColor(int hl) { ... }

void editorSelectSyntaxHighlight() { ... }

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

```

 compiles

First, at the top of the function we make `keywords` an alias for `E.syntax->keywords` since we'll be using it a lot, and in some pretty dense code.

Keywords require a separator both before and after the keyword. Otherwise, the `void` in `avoid`, `voided`, or `avoidable` would be highlighted as a keyword, which is definitely a problem we want to, uh, circumnavigate.

So we check `prev_sep` to make sure a separator came before the keyword, before looping through each possible keyword. For each keyword, we store the length in `klen` and whether it's a secondary keyword in `kw2`, in which case we decrement `klen` to account for the extraneous `|` character.

We then use `strncmp()` to check if the keyword exists at our current position in the text, *and* we check to see if a separator character comes after the keyword. Since `\0` is considered a separator character, this works if the keyword is at the very end of the line.

If all that passed, then we have a keyword to highlight. We use `memset()` to highlight the whole keyword at once, highlighting it with `HL_KEYWORD1` or `HL_KEYWORD2` depending on the value of `kw2`. We then consume the entire keyword by incrementing `i` by the length of the keyword. Then we `break` instead of `continuing`, because we are in an inner loop, so we have to break out of that loop before `continuing` the outer loop. That is why, after the `for` loop, we check if the loop was broken out of by seeing if it got to the terminating `NULL` value, and if it was broken out of, we `continue`.

Nonprintable characters

Before we tackle highlighting multi-line comments, let's take a quick break from `editorUpdateSyntax()`.

We're going to display nonprintable characters in a more user-friendly way. Currently, nonprintable characters completely mess up the rendering that our editor does. Just try running `kilo` and passing itself in as an argument. That is, open the `kilo` executable file using `kilo`. And try moving the cursor around, and typing. It's not pretty. Every keypress causes the terminal to ding, because the audible bell character (7) is being printed out. Strings containing terminal escape sequences in our code are being printed out as actual escape sequences, because that's how they're stored in a raw executable.

To prevent all that, we're going to translate nonprintable characters into printable ones. We'll render the alphabetic control characters (`Ctrl-A` = 1, `Ctrl-B` = 2, ..., `Ctrl-Z` = 26) as the capital letters A through Z. We'll also render the 0 byte like a control character. `Ctrl-@` = 0, so we'll render it as an @ sign. Finally, any other nonprintable characters we'll render as a question mark (?). And to differentiate these characters from their printable counterparts, we'll render them using inverted colors (black on white).

kilo.c

Step 175

nonprintables

```
/** includes */
/** defines */
/** data */
/** filetypes */
/** prototypes */
/** terminal */
/** syntax highlighting */
/** row operations */
/** editor operations */
/** file i/o */
/** find */
/** 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;
    char *c = &E.row[filerow].render[E.coloff];
    unsigned char *hl = &E.row[filerow].hl[E.coloff];
    int current_color = -1;
    int j;
    for (j = 0; j < len; j++) {
        if (iscntrl(c[j])) {
            char sym = (c[j] <= 26) ? '@' + c[j] : '?';
            abAppend(ab, "\x1b[7m", 4);
            abAppend(ab, &sym, 1);
            abAppend(ab, "\x1b[m", 3);
        } else if (hl[j] == HL_NORMAL) {
            if (current_color != -1) {
                abAppend(ab, "\x1b[39m", 5);
                current_color = -1;
            }
            abAppend(ab, &c[j], 1);
        } else {
            int color = editorSyntaxToColor(hl[j]);
            if (color != current_color) {
                current_color = color;
                char buf[16];
                int clen = snprintf(buf, sizeof(buf), "\x1b[%dm", color);
                abAppend(ab, buf, clen);
            }
            abAppend(ab, &c[j], 1);
        }
    }
    abAppend(ab, "\x1b[39m", 5);
}

abAppend(ab, "\x1b[K", 3);
abAppend(ab, "\r\n", 2);
}
}

void editorDrawStatusBar(struct abuf *ab) { ... }

void editorDrawMessageBar(struct abuf *ab) { ... }

```

```
void editorRefreshScreen() { ... }

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

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

[↗ compiles](#)

We use `iscntrl()` to check if the current character is a control character. If so, we translate it into a printable character by adding its value to `'@'` (in ASCII, the capital letters of the alphabet come after the `@` character), or using the `'?'` character if it's not in the alphabetic range.

We then use the `<esc>[7m` escape sequence to switch to inverted colors before printing the translated symbol. We use `<esc>[m` to turn off inverted colors again.

Unfortunately, `<esc>[m` turns off *all* text formatting, including colors. So let's print the escape sequence for the current color afterwards.

kilo.c

Step 176

nonprintables-fix-color

```
/** includes */
/** defines */
/** data */
/** filetypes */
/** prototypes */
/** terminal */
/** syntax highlighting */
/** row operations */
/** editor operations */
/** file i/o */
/** find */
/** 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;
    char *c = &E.row[filerow].render[E.coloff];
    unsigned char *hl = &E.row[filerow].hl[E.coloff];
    int current_color = -1;
    int j;
    for (j = 0; j < len; j++) {
        if (iscntrl(c[j])) {
            char sym = (c[j] <= 26) ? '@' + c[j] : '?';
            abAppend(ab, "\x1b[7m", 4);
            abAppend(ab, &sym, 1);
            abAppend(ab, "\x1b[m", 3);
            if (current_color != -1) {
                char buf[16];
                int clen = snprintf(buf, sizeof(buf), "\x1b[%dm", current_color);
                abAppend(ab, buf, clen);
            }
        } else if (hl[j] == HL_NORMAL) {
            if (current_color != -1) {
                abAppend(ab, "\x1b[39m", 5);
                current_color = -1;
            }
            abAppend(ab, &c[j], 1);
        } else {
            int color = editorSyntaxToColor(hl[j]);
            if (color != current_color) {
                current_color = color;
                char buf[16];
                int clen = snprintf(buf, sizeof(buf), "\x1b[%dm", color);
                abAppend(ab, buf, clen);
            }
            abAppend(ab, &c[j], 1);
        }
    }
    abAppend(ab, "\x1b[39m", 5);
}

abAppend(ab, "\x1b[K", 3);
abAppend(ab, "\r\n", 2);
}
}

void editorDrawStatusBar(struct abuf *ab) { ... }

void editorDrawMessageBar(struct abuf *ab) { ... }

```

```

void editorRefreshScreen() { ... }

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

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

```

[↗ compiles](#)

You can test the coloring of nonprintables by pressing `Ctrl-A`, `Ctrl-B`, and so on to insert those control characters into strings or comments, and you should see that they get the same color as the surrounding characters, just inverted.

Colorful multiline comments

Okay, we have one last feature to implement: multi-line comment highlighting. Let's start by adding `HL_MLCOMMENT` to the `editorHighlight` enum.

kilo.c

Step 177

hl-multiline-comments

```

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

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

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

enum editorKey { ... };

enum editorHighlight {
    HL_NORMAL = 0,
    HL_COMMENT,
    HL_MLCOMMENT,
    HL_KEYWORD1,
    HL_KEYWORD2,
    HL_STRING,
    HL_NUMBER,
    HL_MATCH
};

#define HL_HIGHLIGHT_NUMBERS (1<<0)
#define HL_HIGHLIGHT_STRINGS (1<<1)

/** data */
/** filetypes */
/** prototypes */
/** terminal */
/** syntax highlighting */

int is_separator(int c) { ... }

```

```

void editorUpdateSyntax(erow *row) { ... }

int editorSyntaxToColor(int hl) {
    switch (hl) {
        case HL_COMMENT:
        case HL_MLCOMMENT: return 36;
        case HL_KEYWORD1: return 33;
        case HL_KEYWORD2: return 32;
        case HL_STRING: return 35;
        case HL_NUMBER: return 31;
        case HL_MATCH: return 34;
        default: return 37;
    }
}

void editorSelectSyntaxHighlight() { ... }

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

```

[↪](#) compiles, but with no observable effects

We'll highlight multi-line comments to be the same color as single-line comments (cyan).

Now we'll add two strings to `editorSyntax`: `multiline_comment_start` and `multiline_comment_end`. In C, these will be `"/*"` and `"*/"`.

kilo.c

Step 178

mcs-mce

```

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

struct editorSyntax {
    char *filetype;
    char **filematch;
    char **keywords;
    char *singleline_comment_start;
    char *multiline_comment_start;
    char *multiline_comment_end;
    int flags;
};

typedef struct erow { ... } erow;

struct editorConfig { ... };

```

```

struct editorConfig E;

/** filetypes */

char *C_HL_extensions[] = { ".c", ".h", ".cpp", NULL };
char *C_HL_keywords[] = { ... };

struct editorSyntax HLDB[] = {
    {
        "c",
        C_HL_extensions,
        C_HL_keywords,
        "//", "/*", "*/",
        HL_HIGHLIGHT_NUMBERS | HL_HIGHLIGHT_STRINGS
    },
};

#define HLDB_ENTRIES (sizeof(HLDB) / sizeof(HLDB[0]))

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

```

[↗](#) compiles, but with no observable effects

Now let's open `editorUpdateSyntax()` up once again. We'll add `mcs` and `mce` aliases that are analogous to the `scs` alias we already have for single-line comments. We'll also add `mcs_len` and `mce_len`.

kilo.c

Step 179

mcs-mce-len

```

/** includes */
/** defines */
/** data */
/** filetypes */
/** prototypes */
/** terminal */
/** syntax highlighting */

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) {
    row->hl = realloc(row->hl, row->rsz);
    memset(row->hl, HL_NORMAL, row->rsz);
}

```

```

if (E.syntax == NULL) return;

char **keywords = E.syntax->keywords;

char *scs = E.syntax->singleline_comment_start;
char *mcs = E.syntax->multiline_comment_start;
char *mce = E.syntax->multiline_comment_end;

int scs_len = scs ? strlen(scs) : 0;
int mcs_len = mcs ? strlen(mcs) : 0;
int mce_len = mce ? strlen(mce) : 0;

int prev_sep = 1;
int in_string = 0;

int i = 0;
while (i < row->rsz) {
    char c = row->render[i];
    unsigned char prev_hl = (i > 0) ? row->hl[i - 1] : HL_NORMAL;

    if (scs_len && !in_string) {
        if (!strncmp(&row->render[i], scs, scs_len)) {
            memset(&row->hl[i], HL_COMMENT, row->rsz - i);
            break;
        }
    }

    if (E.syntax->flags & HL_HIGHLIGHT_STRINGS) {
        if (in_string) {
            row->hl[i] = HL_STRING;
            if (c == '\\') && i + 1 < row->rsz) {
                row->hl[i + 1] = HL_STRING;
                i += 2;
                continue;
            }
            if (c == in_string) in_string = 0;
            i++;
            prev_sep = 1;
            continue;
        } else {
            if (c == '"' || c == '\\') {
                in_string = c;
                row->hl[i] = HL_STRING;
                i++;
                continue;
            }
        }
    }

    if (E.syntax->flags & HL_HIGHLIGHT_NUMBERS) {
        if ((isdigit(c) && (prev_sep || prev_hl == HL_NUMBER)) ||
            (c == '.' && prev_hl == HL_NUMBER)) {
            row->hl[i] = HL_NUMBER;
        }
    }
}

```

```

        i++;
        prev_sep = 0;
        continue;
    }
}

if (prev_sep) {
    int j;
    for (j = 0; keywords[j]; j++) {
        int klen = strlen(keywords[j]);
        int kw2 = keywords[j][klen - 1] == '|';
        if (kw2) klen--;

        if (!strcmp(&row->render[i], keywords[j], klen) &&
            is_separator(row->render[i + klen])) {
            memset(&row->hl[i], kw2 ? HL_KEYWORD2 : HL_KEYWORD1, klen);
            i += klen;
            break;
        }
    }
    if (keywords[j] != NULL) {
        prev_sep = 0;
        continue;
    }
}

prev_sep = is_separator(c);
i++;
}
}

```

```
int editorSyntaxToColor(int hl) { ... }
```

```
void editorSelectSyntaxHighlight() { ... }
```

```

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

```

[↗](#) compiles, but with no observable effects

Now for the highlighting code. We won't worry about multiple lines just yet.

kilo.c

Step 180

syntax-mlcomment

```

/**** includes ****/
/**** defines ****/
/**** data ****/
/**** filetypes ****/

```

```

/**/ prototypes */
/**/ terminal */
/**/ syntax highlighting */

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) {
    row->hl = realloc(row->hl, row->rsize);
    memset(row->hl, HL_NORMAL, row->rsize);

    if (E.syntax == NULL) return;

    char **keywords = E.syntax->keywords;

    char *scs = E.syntax->singleline_comment_start;
    char *mcs = E.syntax->multiline_comment_start;
    char *mce = E.syntax->multiline_comment_end;

    int scs_len = scs ? strlen(scs) : 0;
    int mcs_len = mcs ? strlen(mcs) : 0;
    int mce_len = mce ? strlen(mce) : 0;

    int prev_sep = 1;
    int in_string = 0;
    int in_comment = 0;

    int i = 0;
    while (i < row->rsize) {
        char c = row->render[i];
        unsigned char prev_hl = (i > 0) ? row->hl[i - 1] : HL_NORMAL;

        if (scs_len && !in_string) {
            if (!strcmp(&row->render[i], scs, scs_len)) {
                memset(&row->hl[i], HL_COMMENT, row->rsize - i);
                break;
            }
        }

        if (mcs_len && mce_len && !in_string) {
            if (in_comment) {
                row->hl[i] = HL_MLCOMMENT;
                if (!strcmp(&row->render[i], mce, mce_len)) {
                    memset(&row->hl[i], HL_MLCOMMENT, mce_len);
                    i += mce_len;
                    in_comment = 0;
                    prev_sep = 1;
                    continue;
                } else {
                    i++;
                    continue;
                }
            } else if (!strcmp(&row->render[i], mcs, mcs_len)) {
                memset(&row->hl[i], HL_MLCOMMENT, mcs_len);

```

```

        i += mcs_len;
        in_comment = 1;
        continue;
    }
}

if (E.syntax->flags & HL_HIGHLIGHT_STRINGS) {
    if (in_string) {
        row->hl[i] = HL_STRING;
        if (c == '\\\' && i + 1 < row->rsz) {
            row->hl[i + 1] = HL_STRING;
            i += 2;
            continue;
        }
        if (c == in_string) in_string = 0;
        i++;
        prev_sep = 1;
        continue;
    } else {
        if (c == '"' || c == '\') {
            in_string = c;
            row->hl[i] = HL_STRING;
            i++;
            continue;
        }
    }
}

if (E.syntax->flags & HL_HIGHLIGHT_NUMBERS) {
    if ((isdigit(c) && (prev_sep || prev_hl == HL_NUMBER)) ||
        (c == '.' && prev_hl == HL_NUMBER)) {
        row->hl[i] = HL_NUMBER;
        i++;
        prev_sep = 0;
        continue;
    }
}

if (prev_sep) {
    int j;
    for (j = 0; keywords[j]; j++) {
        int klen = strlen(keywords[j]);
        int kw2 = keywords[j][klen - 1] == '|';
        if (kw2) klen--;

        if (!strncmp(&row->render[i], keywords[j], klen) &&
            is_separator(row->render[i + klen])) {
            memset(&row->hl[i], kw2 ? HL_KEYWORD2 : HL_KEYWORD1, klen);
            i += klen;
            break;
        }
    }
}

if (keywords[j] != NULL) {

```

```

        prev_sep = 0;
        continue;
    }
}

prev_sep = is_separator(c);
i++;
}
}

int editorSyntaxToColor(int hl) { ... }

void editorSelectSyntaxHighlight() { ... }

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

```

[↗ compiles](#)

First we add an `in_comment` boolean variable to keep track of whether we're currently inside a multi-line comment (this variable isn't used for single-line comments).

Moving down into the `while` loop, we require both `mcs` and `mce` to be non-NULL strings of length greater than 0 in order to turn on multi-line comment highlighting. We also check to make sure we're not in a string, because having `/*` inside a string doesn't start a comment in most languages. Okay, I'll say it: *all* languages.

If we're currently in a multi-line comment, then we can safely highlight the current character with `HL_MLCOMMENT`. Then we check if we're at the end of a multi-line comment by using `strncmp()` with `mce`. If so, we use `memset()` to highlight the whole `mce` string with `HL_MLCOMMENT`, and then we consume it. If we're not at the end of the comment, we simply consume the current character which we already highlighted.

If we're not currently in a multi-line comment, then we use `strncmp()` with `mcs` to check if we're at the beginning of a multi-line comment. If so, we use `memset()` to highlight the whole `mcs` string with `HL_MLCOMMENT`, set `in_comment` to true, and consume the whole `mcs` string.

Now let's fix a bit of a complication that multi-line comments add: single-line comments should not be recognized inside multi-line comments.

```

/**** includes ****/
/**** defines ****/
/**** data ****/
/**** filetypes ****/
/**** prototypes ****/
/**** terminal ****/
/**** syntax highlighting ****/

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) {
    row->hl = realloc(row->hl, row->rsize);
    memset(row->hl, HL_NORMAL, row->rsize);

    if (E.syntax == NULL) return;

    char **keywords = E.syntax->keywords;

    char *scs = E.syntax->singleline_comment_start;
    char *mcs = E.syntax->multiline_comment_start;
    char *mce = E.syntax->multiline_comment_end;

    int scs_len = scs ? strlen(scs) : 0;
    int mcs_len = mcs ? strlen(mcs) : 0;
    int mce_len = mce ? strlen(mce) : 0;

    int prev_sep = 1;
    int in_string = 0;
    int in_comment = 0;

    int i = 0;
    while (i < row->rsize) {
        char c = row->render[i];
        unsigned char prev_hl = (i > 0) ? row->hl[i - 1] : HL_NORMAL;

        if (scs_len && !in_string && !in_comment) {
            if (!strncmp(&row->render[i], scs, scs_len)) {
                memset(&row->hl[i], HL_COMMENT, row->rsize - i);
                break;
            }
        }

        if (mcs_len && mce_len && !in_string) {
            if (in_comment) {
                row->hl[i] = HL_MLCOMMENT;
                if (!strncmp(&row->render[i], mce, mce_len)) {
                    memset(&row->hl[i], HL_MLCOMMENT, mce_len);
                    i += mce_len;
                    in_comment = 0;
                    prev_sep = 1;
                    continue;
                } else {
                    i++;
                }
            }
        }
    }
}

```

```

        continue;
    }
} else if (!strncmp(&row->render[i], mcs, mcs_len)) {
    memset(&row->hl[i], HL_MLCOMMENT, mcs_len);
    i += mcs_len;
    in_comment = 1;
    continue;
}
}

if (E.syntax->flags & HL_HIGHLIGHT_STRINGS) {
    if (in_string) {
        row->hl[i] = HL_STRING;
        if (c == '\\\ ' && i + 1 < row->rsiz) {
            row->hl[i + 1] = HL_STRING;
            i += 2;
            continue;
        }
        if (c == in_string) in_string = 0;
        i++;
        prev_sep = 1;
        continue;
    } else {
        if (c == '"' || c == '\\\'') {
            in_string = c;
            row->hl[i] = HL_STRING;
            i++;
            continue;
        }
    }
}

if (E.syntax->flags & HL_HIGHLIGHT_NUMBERS) {
    if ((isdigit(c) && (prev_sep || prev_hl == HL_NUMBER)) ||
        (c == '.' && prev_hl == HL_NUMBER)) {
        row->hl[i] = HL_NUMBER;
        i++;
        prev_sep = 0;
        continue;
    }
}

if (prev_sep) {
    int j;
    for (j = 0; keywords[j]; j++) {
        int klen = strlen(keywords[j]);
        int kw2 = keywords[j][klen - 1] == '|';
        if (kw2) klen--;

        if (!strncmp(&row->render[i], keywords[j], klen) &&
            is_separator(row->render[i + klen])) {
            memset(&row->hl[i], kw2 ? HL_KEYWORD2 : HL_KEYWORD1, klen);
            i += klen;
        }
    }
}

```

```

        break;
    }
}
if (keywords[j] != NULL) {
    prev_sep = 0;
    continue;
}
}

prev_sep = is_separator(c);
i++;
}
}

int editorSyntaxToColor(int hl) { ... }

void editorSelectSyntaxHighlight() { ... }

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

```

[↪](#) compiles, but with no observable effects

Okay, now let's work on highlighting multi-line comments that actually span over multiple lines. To do this, we need to know if the previous line is part of an unclosed multi-line comment. Let's add an `hl_open_comment` boolean variable to the `erow` struct. Let's also add an `idx` integer variable, so that each `erow` knows its own index within the file. That will allow each row to examine the previous row's `hl_open_comment` value.

kilo.c

Step 182

idx-and-hloc

```

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

struct editorSyntax { ... };

typedef struct erow {
    int idx;
    int size;
    int rsize;
    char *chars;
    char *render;
    unsigned char *hl;
    int hl_open_comment;
} erow;

```

```

struct editorConfig { ... };

struct editorConfig E;

/**/ filetypes ***/
/**/ prototypes ***/
/**/ terminal ***/
/**/ syntax highlighting ***/
/**/ row operations ***/

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

int editorRowRxToCx(erow *row, int rx) { ... }

void editorUpdateRow(erow *row) { ... }

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

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

    E.row[at].idx = at;

    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.row[at].hl = NULL;
    E.row[at].hl_open_comment = 0;
    editorUpdateRow(&E.row[at]);

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

void editorFreeRow(erow *row) { ... }

void editorDelRow(int at) { ... }

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

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

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

/**/ editor operations ***/
/**/ file i/o ***/
/**/ find ***/

```

```

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

```

[↗](#) compiles, but with no observable effects

We initialize `idx` to the row's index in the file at the time it is inserted. Let's make sure to update the `idx` of each row whenever a row is inserted into or removed from the file.

kilo.c

Step 183

update-idx

```

/**** includes ****/
/**** defines ****/
/**** data ****/
/**** filetypes ****/
/**** prototypes ****/
/**** terminal ****/
/**** syntax highlighting ****/
/**** row operations ****/

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

int editorRowRxToCx(erow *row, int rx) { ... }

void editorUpdateRow(erow *row) { ... }

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

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

    E.row[at].idx = at;

    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.row[at].hl = NULL;
    E.row[at].hl_open_comment = 0;
    editorUpdateRow(&E.row[at]);

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

void editorFreeRow(erow *row) { ... }

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

[↗](#) compiles, but with no observable effects

The for loops update the index of each row that was displaced by the insert or delete operation.

Now, the final step.

kilo.c

Step 184

propagate-highlight

```

/**/ includes */
/**/ defines */
/**/ data */
/**/ filetypes */
/**/ prototypes */
/**/ terminal */
/**/ syntax highlighting */

int is_separator(int c) { ... }

void editorUpdateSyntax(erow *row) {
    row->hl = realloc(row->hl, row->rsize);
    memset(row->hl, HL_NORMAL, row->rsize);

    if (E.syntax == NULL) return;

    char **keywords = E.syntax->keywords;

    char *scs = E.syntax->singleline_comment_start;
    char *mcs = E.syntax->multiline_comment_start;
    char *mce = E.syntax->multiline_comment_end;

```

```

int scs_len = scs ? strlen(scs) : 0;
int mcs_len = mcs ? strlen(mcs) : 0;
int mce_len = mce ? strlen(mce) : 0;

int prev_sep = 1;
int in_string = 0;
int in_comment = (row->idx > 0 && E.row[row->idx - 1].hl_open_comment);

int i = 0;
while (i < row->rsz) {
    char c = row->render[i];
    unsigned char prev_hl = (i > 0) ? row->hl[i - 1] : HL_NORMAL;

    if (scs_len && !in_string && !in_comment) {
        if (!strncmp(&row->render[i], scs, scs_len)) {
            memset(&row->hl[i], HL_COMMENT, row->rsz - i);
            break;
        }
    }

    if (mcs_len && mce_len && !in_string) {
        if (in_comment) {
            row->hl[i] = HL_MLCOMMENT;
            if (!strncmp(&row->render[i], mce, mce_len)) {
                memset(&row->hl[i], HL_MLCOMMENT, mce_len);
                i += mce_len;
                in_comment = 0;
                prev_sep = 1;
                continue;
            } else {
                i++;
                continue;
            }
        } else if (!strncmp(&row->render[i], mcs, mcs_len)) {
            memset(&row->hl[i], HL_MLCOMMENT, mcs_len);
            i += mcs_len;
            in_comment = 1;
            continue;
        }
    }

    if (E.syntax->flags & HL_HIGHLIGHT_STRINGS) {
        if (in_string) {
            row->hl[i] = HL_STRING;
            if (c == '\\\\' && i + 1 < row->rsz) {
                row->hl[i + 1] = HL_STRING;
                i += 2;
                continue;
            }
        }
        if (c == in_string) in_string = 0;
        i++;
        prev_sep = 1;
        continue;
    }
}

```

```

    } else {
        if (c == '"' || c == '\\') {
            in_string = c;
            row->hl[i] = HL_STRING;
            i++;
            continue;
        }
    }
}

if (E.syntax->flags & HL_HIGHLIGHT_NUMBERS) {
    if ((isdigit(c) && (prev_sep || prev_hl == HL_NUMBER)) ||
        (c == '.' && prev_hl == HL_NUMBER)) {
        row->hl[i] = HL_NUMBER;
        i++;
        prev_sep = 0;
        continue;
    }
}

if (prev_sep) {
    int j;
    for (j = 0; keywords[j]; j++) {
        int klen = strlen(keywords[j]);
        int kw2 = keywords[j][klen - 1] == '|';
        if (kw2) klen--;

        if (!strncmp(&row->render[i], keywords[j], klen) &&
            is_separator(row->render[i + klen])) {
            memset(&row->hl[i], kw2 ? HL_KEYWORD2 : HL_KEYWORD1, klen);
            i += klen;
            break;
        }
    }
    if (keywords[j] != NULL) {
        prev_sep = 0;
        continue;
    }
}

prev_sep = is_separator(c);
i++;
}

int changed = (row->hl_open_comment != in_comment);
row->hl_open_comment = in_comment;
if (changed && row->idx + 1 < E.numrows)
    editorUpdateSyntax(&E.row[row->idx + 1]);
}

int editorSyntaxToColor(int hl) { ... }

void editorSelectSyntaxHighlight() { ... }

```

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

 [compiles](#)

Near the top of `editorUpdateSyntax()`, we initialize `in_comment` to true if the previous row has an unclosed multi-line comment. If that's the case, then the current row will start out being highlighted as a multi-line comment.

At the bottom of `editorUpdateSyntax()`, we set the value of the current row's `hl_open_comment` to whatever state `in_comment` got left in after processing the entire row. That tells us whether the row ended as an unclosed multi-line comment or not.

Then we have to consider updating the syntax of the next lines in the file. So far, we have only been updating the syntax of a line when the user changes that specific line. But with multi-line comments, a user could comment out an entire file just by changing one line. So it seems like we need to update the syntax of all the lines following the current line. However, we know the highlighting of the next line will not change if the value of this line's `hl_open_comment` did not change. So we check if it changed, and only call `editorUpdateSyntax()` on the next line if `hl_open_comment` changed (and if there is a next line in the file). Because `editorUpdateSyntax()` keeps calling itself with the next line, the change will continue to propagate to more and more lines until one of them is unchanged, at which point we know that all the lines after that one must be unchanged as well.

You're done

That's it! Our text editor is finished. In the [appendices](#), you'll find some ideas for features you might want to extend the editor with on your own.

[1.0.0beta11](#) ([changelog](#))

[top of page](#)

Appendices

How the diffs work

Each step in this tutorial is presented as a diff. A diff shows you the changes you need to make to the previous step's code to get to the current step. Here's a sample diff, from step 7:

kilo.c	Step 7	icanon
<pre>#include <stdlib.h> #include <termios.h> #include <unistd.h> struct termios orig_termios; void disableRawMode() { ... } void enableRawMode() { tcgetattr(STDIN_FILENO, &orig_termios); atexit(disableRawMode); struct termios raw = orig_termios; raw.c_lflag &= ~(ECHO ICANON); tcsetattr(STDIN_FILENO, TCSAFLUSH, &raw); } int main() { ... }</pre>		
		↗ compiles

Each diff starts with a header that contains the filename of the file you need to edit (“kilo.c”), the step number (“Step 7”), and the step name (“icanon”). You can click the filename to see the full source code of the file for the current step on GitHub. You can also click the step name on the far right to browse all files for the current step on GitHub (which isn’t particularly useful for this tutorial, since we’re just working on a single source file).

After the header, the contents of the file are shown. Lines that need to be added or changed are highlighted and marked with an arrow. Functions that don’t contain any changed code are folded into a single line with their contents hidden.

Lines that need to be removed are given a red background, a ~~strike-through~~ style, and are marked with an ✕. Removed lines are not shown when they are adjacent to an added or changed line, so you won’t see them very often.

The bottom of each diff shows you the compile status of that step. If it's green and says "compiles", then you can expect your code to compile after completing the step, and you can expect to be able to observe the change when you run the program. If there are no observable changes for that step, then the compile status will be blue and say, "compiles, but with no observable effects". On the rare occasion that the step doesn't compile, it will be red and say "doesn't compile".

What to do if you are stuck

Some of the code in this tutorial is very tricky to type in exactly, especially if you're not used to C. It's especially easy to make a mistake when you're making a change to a line, and you think you're done changing that line, but you missed one little change to another part of that same line. It's important to take your time, and compare the changed parts of the diff *character-by-character* with your code to make sure they're the same.

If you suspect you made an error, but don't know where it is or how far back you might've made the error, you should get your computer to do a diff between your version of `kilo.c` and the tutorial's version of `kilo.c` for whatever step you're on. The [kilo-src](#) repository contains the `kilo.c` source code for every step in the tutorial.

You will need `git` to do this. To install `git` (assuming you've completed [chapter 1](#)): on **Ubuntu/Bash on Windows**, run `sudo apt-get install git`; on **Cygwin**, run the installer again and select the `git` package for installation; on **macOS**, `git` should've been installed when you installed command line tools.

Once you have `git` installed, clone the [kilo-src](#) repository by running `git clone https://github.com/snaptoken/kilo-src`. `cd` into the repo using `cd kilo-src`. The repo has a [tag](#) for each step that points the step name to that step's commit in the repo. So to get the source code for the step named `icanon`, run `git checkout icanon`. The `kilo.c` file will now contain the code for that step. You can compare your `kilo.c` with this `kilo.c` by running something like `git diff --no-index -b ../path/to/your/kilo.c kilo.c`. This will show you the changes you would need to make to your `kilo.c` to get it to look like the one in the repo. The `-b` option ignores whitespace, so it won't matter if you use a different indent style than the one in the tutorial.

Where to get help

If you are having trouble, feel free to create an [issue](#) on the tutorial's [GitHub repo](#), and ask a question.

You can also [email me](#) directly if you'd rather not use GitHub.

Ideas for features to add on your own

If you want to extend `kilo` on your own, I suggest trying to actually *use* `kilo` as your text editor for a while. You will very quickly become painfully aware of all sorts of features you're used to having in a text editor, but are missing in `kilo`. Those are the features you should try to add. And you should use `kilo` when you work on `kilo.c`.

If you're still looking for ideas, here's a small list, roughly in order of increasing difficulty.

- **More filetypes:** Add syntax highlighting rules for some of your favourite languages to the HLDB array.
- **Line numbers:** Display the line number to the left of each line of the file.
- **Soft indent:** If you like using spaces instead of tabs, make the `Tab` key insert spaces instead of `\t`. You may want `Backspace` to remove a `Tab` key's worth of spaces as well.
- **Auto indent:** When starting a new line, indent it to the same level as the previous line.
- **Hard-wrap lines:** Insert a newline in the text when the user is about to type past the end of the screen. Try not to insert the newline where it would split up a word.
- **Soft-wrap lines:** When a line is longer than the screen width, use multiple lines on the screen to display it instead of horizontal scrolling.
- **Use ncurses:** The [ncurses](#) library takes care of a lot of the low level terminal interaction for you, and makes your program more portable.
- **Copy and paste:** Give the user a way to select text, and then copy the selected text when they press `Ctrl-C`, and let them paste the copied text when they press `Ctrl-V`.
- **Config file:** Have `kilo` read a config file (maybe named `.kilorc`) to set options that are currently constants, like `KILO_TAB_STOP` and `KILO_QUIT_TIMES`. Try to make more things configurable.
- **Modal editing:** If you like [vim](#), make `kilo` work more like vim by letting the user press `i` for "insert mode" and then press `Escape` to go back to "normal mode". Then start adding all your favourite vim commands, starting with the basic movement commands (`h` `j` `k` `l`).
- **Multiple buffers:** Allow having multiple files open at once, and have some way of switching between them.

More tutorials like this

I am planning to make more tutorials like this one. They will all be available at viewsourcecode.org/snaptoken. There is a link there that will let you sign up to receive

an email whenever a new tutorial is available. There is also a list of similar tutorials by other people from around the web.

The next tutorials will be a little different from this one. For example, one might be a [password manager](#) in 700 lines of shell script, and another might be a [web microframework](#) implemented as just a big rectangle of obfuscated Ruby.

What the tutorials will have in common is the step-by-step build-it-yourself approach to reading and understanding the code of real open-source software projects. If there was a toy like Lego that involved putting *programs* together instead of physical structures, I think “snaptoken” would be a great name for it. That is the experience I’m trying to create with tutorials like this.

How to contribute

Contributions are welcome, whether it’s changes to the text, the code, or the HTML/CSS.

The text is in the `doc/` directory of the [kilo-tutorial](#) repo. Each chapter is a markdown (`.md`) file.

The HTML/CSS is in the `doc/html_in/` directory.

The code is in `steps.diff`, which isn’t human-editable. It is generated by a program called [leg](#).

If you are making significant changes to the text, you probably want to generate the final static HTML files, to preview your changes. Here is how to generate the HTML output using the `leg` program:

1. You need to have [Ruby](#) installed.
2. Install the `leg` binary by running `gem install snaptoken` (you may need to `sudo` this).
3. Inside the `kilo-tutorial` repo, run `leg doc` to generate the static HTML files in `doc/html_out/` and `doc/html_offline/`.
4. When running `leg doc`, each step’s diff is cached in a hidden dotfile, so as long as you’re only making changes to files in the `doc/` folder, you can run `leg doc -c` to use the cached diffs and regenerate the HTML output way faster.

If you just have a small correction to make in the text, there is no need to go through all this. Just make the change in the chapter’s markdown file and [submit a pull request](#).

Credits

[antirez](#) is the author of [kilo](#). He wrote a [blog post](#) about it, in which he explains how he reused code from two of his other projects to quickly throw together `kilo` in just a few hours during a couple already busy weekends. It's not the sort of pristine code you usually see in programming tutorials, but I like it this way. I originally intended this tutorial to be an experimental form of documentation for his code, until I started making changes to the code all over the place to make for a better reading experience.

I used many of the [patches](#) submitted to the `kilo` GitHub page to fix various bugs in `kilo`. The [openemacs](#) project (a fork of `kilo`) was also helpful as a reference.

I used [redcarpet](#) to render the Markdown source of this tutorial to HTML, and I used [rouge](#) for syntax highlighting.

If you want to know more about me, see [viewsourcocode.org](#).

License

The `kilo` source code is released under the [BSD 2-Clause](#) license.

The rest of the tutorial is licensed under [CC BY 4.0](#).

1.0.0beta11 ([changelog](#))

[top of page](#)