← prev contents next →

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.c **Step 3** read

```
#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 `00000000000000000000000000001000` in binary. We use the bitwise-NOT operator (`~`) on this value to get `11111111111111111111111111110111`. 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;
}
```

↗ compiles

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

`iscntrl()` 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.c **Step 9** isig

```
#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.c **Step 10** `ixon`

```
#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.

kilo.c **Step 14** `carriage-returns`

```
#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;
}
```

↗ 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.

kilo.c **Step 15** `misc-flags`

```
#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);

  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.

kilo.c **Step 16** vmin-vtime

```
#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 `0`s 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.

kilo.c **Step 17** `die`

```
#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() { … }
```

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