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

```
#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

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 viewsourcecode.org.

License