← prev contents next →

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