← prev contents next →

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 `rsize` 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--) 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.

kilo.c **Step 151** search-highlighting

```
/*** 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->rsize);
      memcpy(saved_hl, row->hl, row->rsize);
      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.

kilo.c **Step 154** `is-separator`

```
/*** 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.

kilo.c **Step 155** `prev-sep`

```
/*** 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->rsize);
  memset(row->hl, HL_NORMAL, row->rsize);

  if (E.syntax == NULL) return;

  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 (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()`ting 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 `continue`ing 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.

kilo.c **Step 168** `string-escapes`

```
/*** 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 `continue`ing, because we are in an inner loop, so we have to break out of that loop before `continue`ing 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->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 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, 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 (!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->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 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.

kilo.c **Step 181** `slcomment-within-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 && !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->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, 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->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->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 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