Login Interface Documentation

This documentation will take you through the Login Interface code line by line and explain each instruction in detail.

1 `main.c` file

Starting with the `main.c` file—the starting point of all C programs—we have the following code inside the `main()` function:

Listing 1: `main.c`

```c
int main()
{
    /* CURSES INIT */
    initscr();
    noecho();
    keypad(stdscr, TRUE);

    /* INITIATE LOGIN INTERFACE */
    init_login();

    endwin();

    return 0;
}
```

As you can see, the very first thing that was—and should be—done was calling `initscr()`. This function must be called at the beginning of every program using the `<curses.h>` library to initialize the library before using any other curses functions.

Next up are the functions `noecho()`, `keypad()` and `endwin()`.

- `noecho()` disables the automatic echoing of characters typed by the user on the console. By default, when a user types a character, it gets echoed back to the terminal. When using `noecho()`, the user's input won't be visible on the screen. This is useful when you want to make a custom terminal application.
- `keypad(stdscr, TRUE)` enables the keypad of the terminal, which refers to a set of keys on some terminals that include function keys and arrow keys. By default, these keys are treated as normal characters, but when `keypad(stdscr, TRUE)` is called, we are basically telling curses to return them as a set of special codes to process them. Here, `stdscr` refers to the standard screen (the terminal) in curses, and `TRUE` is a constant defined in `<curses.h>` to have a value of 1, which tells curses to enable the keypad (to set it to `TRUE`).
- `endwin()` is used to restore the terminal settings to their original state. It performs a series of cleanup operations on the screen, such as releasing any memory used by the curses library, flushing the output buffer, and reverting any changes made to the cursor. It is important to call this routine at the end of every curses program.

Lastly, we have the function `init_login()`, which initiates the login user interface. It is declared in the `login.h` header file located in the `include` folder. I will go through what this function does in the upcoming sections.

2 Utilities

Before we begin anything, we first need to define a set of utilities to make our lives easier. I put them in a separate header file called `utils.h` inside the `include` folder. All header files are going to be inside this folder, whereas `.c` source files are going to be kept inside the `src` folder to keep things organized.

You may have noticed that I didn't call the `<curses.h>` and `"utils.h"` header files in `main.c`. That is because I already called `<curses.h>` inside `utils.h`. And since we are going to be using `"utils.h"` in literally every other header file, we don't need to call it again. Though calling a header file more than once won't affect the compilation process, mainly because of the include guards (those nifty `#ifndef`, `#define` and `#endif` at the very top and bottom), I just don't like redundant `#include`s.

Now, starting with macros, we have the following macros:

Listing 2: `utils.h`

```c
#define GROW_CAPACITY(cap) \
    (cap) * 2
#define GROW_ARRAY(type, arr, cap) \
    (type *)reallocate(arr, GROW_CAPACITY(cap) * sizeof(type))
```

If you don't know what macros are, a macro is a fragment of code that can be defined once and reused multiple times throughout a program. They are defined using the `#define` preprocessor directive and replaced with their expanded code during compilation. The backslash at the end of the line tells the preprocessor that the macro continues to the next line. This is just to make the code more readable and organized.

Taking the first macro definition as an example, if we use it like this:

```c
int cap = GROW_CAPACITY(2);
```

then, the preprocessor will replace `GROW_CAPACITY(2)` with the expanded code `(2) * 2`, which will be evaluated as `4` by the compiler.

There are other types of macros, but for now, we will only be using these function-like macros.

Both of these two macros are going to be useful when dealing with dynamic arrays. For each dynamic array, we need to keep track of both the length of the array and its capacity. We could keep track of the length only and increase the size of the array by `1` each time we need to add a new element, but that's going to affect performance negatively. For this reason, we are going to start with an array of fixed length, say 4, and double that capacity each time we run out of slots.

This is exactly what the first macro is for; it helps us calculate the new capacity without having to hardcode the formula whenever we need it. This makes it easier to modify the the code later.

As for the second macro, it basically reallocates more memory for the array, and returns a pointer to the new array. It is useful because we won't need to type that whole line; we can actually pass the type of the array as an argument of the macro as well.

You can literally (probably) pass anything as an argument in macros—even symbols (as long as they are not `(`, `)` or `,`). Aren't macros just really cool?

Here is the source code for the function `reallocate()` called by the macro:

Listing 3: `utils.c`

```c
void * reallocate(void * arr, size_t new_cap)
{
    void * new_arr = realloc(arr, new_cap);
    if (!new_arr) exit(1);

    return new_arr;
}
```

If this is the first time you're seeing a pointer of type `void`, don't worry, it's not that complicated. Since we want this function to work for any type of array, we won't specify a type. Therefore, it will be of type `void *`, so that when the function returns the pointer, we can just cast back its original type and convert it from `void *`. A `void` pointer is basically a pointer without a specific type assigned to it. This is a technique used in C by many built-in functions like `malloc()`, `realloc()`,...etc.

So, what this function does is that it allocates more memory for the array. If it succeeds, it returns a pointer to it. Otherwise, it just throws an error.
To be able to use this function outside the scope of `utils.c`, we just add its prototype to the `utils.h` header file and include that header file in `utils.c`. The compiler should do its job and link the files to the `main.c` file.

Moving on to the next set of utility functions:

Listing 4: `utils.h`

```c
WINDOW * create_newwin(int height, int width, int starty, int startx, int type);
void destroy_win(WINDOW *);

void ref_mvwaddch(WINDOW *, int starty, int startx, wchar_t);
```

The first one helps us create a new window with a specific height, width and type, at a specific location on the screen. There are two types of windows here: one with a single border, and one with a double border (these are just decorations).

Now, windows in this context are basically portions created on the screen (`stdscr`) to organize the user interface. Here is how the first one is implemented:

Listing 5: `utils.c`

```c
// Create a new window at the specified position
WINDOW * create_newwin(int height, int width, int starty, int startx, int type)
{
    // Create a new window object
    WINDOW * win = newwin(height, width, starty, startx);
    // Create a box around the window
    if (type == 0)
        // 0, 0 gives default characters for vertical and horizontal lines
        box(win, 0, 0);
    else
        // Double border
        wborder(win,
                186, // left
                186, // right
                205, // top
                205, // bottom
                201, // top left
                187, // top right
                200, // bottom left
                188  // bottom right
                );
    keypad(win, TRUE);
    wrefresh(win);
    return win;
}
```

We first create a new window using the function `newwin()` that comes with <curses.h>. It returns a pointer to the newly created window (we don't want to keep copying the window around, obviously). If `type == 0`, then it creates the default border around our screen. Otherwise, we make our own custom double border with special ASCII characters using `wborder()` from <curses.h> (you can look up the ASCII codes above to get a feel of what the window would look like). Then, we enable the terminal keypad for this new window. And lastly, we refresh it to show it on the screen using `wrefresh()` from <curses.h> and return a pointer to it.

Listing 6: `utils.c`

```c
// Destroy a window
void destroy_win(WINDOW * win)
{
    // Clear borders
    wborder(win,
            ' ', // left
            ' ', // right
            ' ', // top
            ' ', // bottom
            ' ', // top left
            ' ', // top right
            ' ', // bottom left
            ' '  // bottom right
            );
    wrefresh(win);
    // Deallocate memory designated to the window
    delwin(win);
}
```

For the function above, it is pretty straightforward. It basically replaces the borders of the window with whitespaces, refreshes to display the changes on the screen, and then frees the memory designated to it using `delwin()` from `<curses.h>`.

The last function is there just for convenience's sake and to save time typing code.

Listing 7: `utils.c`

```c
void ref_mvwaddch(WINDOW * win, int starty, int startx, wchar_t ch)
{
    mvwaddch(win, starty, startx, ch);
    wrefresh(win);
}
```

It serves as a variant of `mvaddch()` from `<curses.h>`. This function moves the cursor to the position `(startx, starty)` on the window `win` and then prints the character `ch` at that location. However, since the changes won't be shown to the screen until we `wrefresh()`, I created this function so that I wouldn't have to add `wrefresh()` every time, and because I didn't feel like making a macro for that.

Regarding that `wchar_t` type, it is just another data type in C declared in `<wchar.h>`. It is used to represent wide characters, which are characters that are outside the ASCII range (`0-127`). In `<curses.h>`, this type is used to represent special key codes because one byte is not enough to hold some special key codes.

3 Basic Structures

Now that we are done with utilities, we can start defining our main structures, which will be the building blocks for our login/registration form.

The first thing we are going to need, obviously, is a `FORM` structure.

Listing 8: `form.h`

```c
// Form structure
typedef struct
{
    short rows; // Size in rows
    short cols; // Size in cols
    short n_fields; // Number of fields
    short n_buttons; // Number of buttons
    FIELD ** fields; // Array of form fields
    BUTTON ** buttons; // Array of buttons
    WINDOW * win; // Form window
} FORM;
```

For each `FORM`, we need to keep track of its height and width. `FORM`s also have their very own windows. Additionally, each form contains a set of text fields and buttons, so we add two dynamic arrays of types `FIELD *` and `BUTTON *` respectively (I made them as arrays of pointers to the fields and buttons to conserve memory) and store the number of `FIELD`s and `BUTTON`s we have.
What are the `FIELD` and `BUTTON` types, you ask? These are custom structures we are going to need throughout our program.
Here are their definitions:

Listing 9: `form.h`

```c
// Field structure
typedef struct
{
    short rows; // Size in rows
    short cols; // Size in cols
    char * buffer; // Text buffer
    int length; // Buffer length (without '\0')
    int capacity; // Buffer capacity
    WINDOW * win; // Field window
    bool hidden;
} FIELD;

// Button structure
typedef struct
{
    short xpos; // x position
    short ypos; // y position
    const char * content; // Text content
    chtype style; // Button default style
    chtype highlight; // Button highlight style
} BUTTON;
```

Each `FIELD` has its own height and width, it also has its own window. To store its content, we need a buffer, which is going to be a dynamic array of type `char *`. For this reason, we will also need to keep track of the length of the string and the capacity of the buffer.
The `hidden` attribute is just a simple `bool` switch to control whether the text is hidden or not. This will be useful for password fields. And yes, `bool` exists in C. Whoever told you it doesn't lied to you. It is in <stdbool.h>.

As for the `BUTTON` structure, it is pretty simple. We only need its coordinates on a specific window (the form window in this case), the content (of type `const char *` because they are not directly modifiable), and lastly, the default style and the highlight style. You don't need to worry about these right now.

For your information, the order of the attributes of each structure is not arbitrary, and was chosen for a minimal memory usage. The order *does* matter here, and it directly affects the `sizeof` structures. You can look up *"Structure Padding and Memory Alignment in C"* for more information.

4 `login.c` file

Now, let's get to the heart of the matter. Starting with the `init_login()` function:

Listing 10: `login.c`

```c
// Initiate the login interface
void init_login(void)
{
    // Initialize and set default form
    refresh();
    g_form = create_loginform();

```

```c
    get_user_input();
}
```

We first create a new form. The default form will be the login form. We are using the `create_loginform()` function from `"form.h"`, which we should include in our `login.c` file. Here is the source code:

Listing 11: `form.h`

```c
FORM * create_loginform(void);
```

Listing 12: `form.c`

```c
// Create a login form
FORM * create_loginform(void)
{
    // Center the window
    int starty = (LINES - form_height) / 2,
        startx = (COLS - form_width) / 2;

    // Init form
    FORM * form = new_form(form_height, form_width, starty, startx, 1);

    mvwprintw(form->win, 2, (form->cols - 5) / 2, "Login");

    // Allocate memory for two text fields
    form->n_fields = 2;
    init_fields(&form->fields, form->n_fields);
    // Allocate memory for two buttons
    form->n_buttons = 2;
    init_buttons(&form->buttons, form->n_buttons);

    short field_width = form_width - 12;
    short field_height = 3;
    short padding = (form_width - field_width) / 2;

    // First field (email)
    add_field(form, form->fields[0], " - Email Address :", field_height, field_width, 7, false);
    // Second field (password)
    add_field(form, form->fields[1], " - Password :", field_height, field_width, 12, true);

    // First button (submit)
    add_button(form, form->buttons[0], "  Submit  ", 17, padding + 2);
    // Second button (register)
    mvwprintw(form->win, 19, padding - 1, " Don't have an account?");
    add_button(form, form->buttons[1], "  Register  ", 21, padding + 2);
    wrefresh(form->win);

    return form;
}
```

We calculate the coordinates of the top left corner in such a way that the window will be perfectly centered, then we create a new form and initialize its fields using the `new_form()` utility function. This is a `static` function, meaning that it cannot be used outside the scope of this file.

If you are wondering what `form_height` and `form_width` are, they are constants I defined in the `form.h` header file.

Listing 13: `form.h`

```c
#define form_height 27
#define form_width 54
```

Listing 14: `form.c`

```c
// Create an initial form
static FORM * new_form(int height, int width, int starty, int startx, int type)
{
    // Allocate memory for the form
    FORM * form = (FORM *)malloc(sizeof(FORM));
    // Init form window
    form->win = create_newwin(height, width, starty, startx, type);

    form->rows = height;
    form->cols = width;

    form->fields = NULL;
    form->buttons = NULL;

    return form;
}
```

This function first allocates some memory for the new form, creates and initialzes a new window for it using `create_newwin()` utility function I described before, and then initializes its attributes, and returns a pointer to it.

Going back to our lovely `create_login()` function, after we create a new empty form and set a pointer to it (so that we won't have to copy the whole thing), we then print the title `"Login"` at the center of the form window (not `stdscr`) at row 2 (two rows below the border). After that, we add two text fields and two buttons and allocate the needed memory for them using the `init_fields()` and `init_buttons()` functions respectively. Keep in mind that these functions take as arguments the array passed by reference not by value, plus the number of elements. So, we need to pass the address of the array.

Next, we add those fields and buttons to the form window using the two functions `add_field()` and `add_button()` with their respective labels. We print the text `"Don't have an account?"` right before the `"Register"` button, then we `wrefresh()` the form window.
Here is the code for all of those functions I just mentioned:

Listing 15: `form.c`

```c
// Initialize fields
static void init_fields(FIELD *** fields, short n_fields)
{
    *fields = (FIELD **)malloc(n_fields * sizeof(FIELD *));
    for (int i = 0; i < n_fields; i++)
    {
        (*fields)[i] = (FIELD *)malloc(sizeof(FIELD));
        (*fields)[i]->capacity = 4;
        (*fields)[i]->length = 0;
        (*fields)[i]->buffer = (char *)malloc((*fields)[i]->capacity * sizeof(char));
        (*fields)[i]->buffer[0] = '\0';
    }
}

// Initialize buttons
static void init_buttons(BUTTON *** buttons, short n_buttons)
{
    *buttons = (BUTTON **)malloc(n_buttons * sizeof(BUTTON *));
    for (int i = 0; i < n_buttons; i++)
        (*buttons)[i] = (BUTTON *)malloc(sizeof(BUTTON));
}
```

These functions are pretty similar, except that `init_fields()` has more work to do than `init_buttons()`. Here, `init_fields()` starts by allocating `n_fields` number of `FIELD`s, then it loops through the array to allocate memory for each element in the array. It allocates memory for each field in the array, as well as the `buffer`, after initializing the `capacity` and `length`.

Listing 16: `form.c`

```
static void add_field(FORM * form, FIELD * field, const char * label, short height, short width, short ypos, bool hidden)
{
    field->rows = height;
    field->cols = width;
    mvwprintw(form->win, ypos - 2, (form_width - width) / 2, label);
    field->win = create_newwin(height, width, ypos, (COLS - width) / 2, 0);
    field->hidden = hidden;
}

static void add_button(FORM * form, BUTTON * button, const char * content, short ypos, short xpos)
{
    button->xpos = xpos;
    button->ypos = ypos;

    button->content = content;
    button->style = A_BLINK;
    button->highlight = A_STANDOUT;

    wattron(form->win, button->style);
    mvwprintw(form->win, ypos, xpos, button->content);
    wattroff(form->win, button->style);
}
```

The `add_button()` function sets the button coordinates and content to the values passed in the arguments. The new thing here is the `A_BLINK` and `A_STANDOUT` values assigned to `button->style` and `button->highlight`. These are constants/attributes that come with `<curses.h>` that allow us to change to colors and/or styles of our text. The way we do this is by calling `wattron()` on the form window, to enable the attribute we need, then print the button `content` using `mvwprintw()`, before we disable the attribute using `wattroff()`. The `mvwprintw()` function first moves the cursor to the location `(xpos, ypos)` on the form window (these are relative coordinates to the form window), then prints the `content`.
`add_field()` functions the same way, and it is pretty straightforward, so I will skip it for now.

Finally, we give the output of the `create_loginform()` function (which is a pointer to the newly created form) to a global `FORM` pointer called `g_form`. We are going to use this pointer to switch between the login form and the registration form.

Listing 17: `login.c`

```
static FORM * g_form;
```

5 Getting the user input

5.1 Using the Up and Down arrow keys

Now that we have created our stunning and nice-looking login form, we are ready to start getting input from the user. We are going to do this inside the `get_user_input()` function.

The basic idea here is to test for each key pressed by the user to check if it is printable or not. If it is, then we add it to the buffer and print it on the screen. If it isn't, then we check if it is a control key (e.g. Backspace).
For this, we need a new variable `ch`, which is where we are going to store the user input temporarily, and we need it to be large enough to hold special codes (like arrow key codes). `wchar_t` is going to work better than `char` in this case.

Listing 18: `login.c`

```
    wchar_t ch; // Character to scan
```

We start an infinite `while` loop (it is going to be infinite just for now), then scan a character code from our field window and store it in `ch`:

Listing 19: `login.c`

```c
    while (true)
    {
        ch = wgetch(g_form->fields[i]->win);
```

The variable `i` is just going to have a value of 0 for now, meaning that we will be working with the first text field of our form only. So, we need to move the cursor the this field before anything:

Listing 20: `login.c`

```c
    short i = 0; // Index of the selected field/button
```

Listing 21: `login.c`

```c
    // Show the cursor if hidden, then move it to the beginning of the first field
    curs_set(1);
    wmove(g_form->fields[0]->win, 1, 1);
    wrefresh(g_form->fields[0]->win);
```

We set the visibility of the cursor to 1 using `curs_set(1)`. Then, we move it to the location `(1, 1)` of the first field in our form (note that these coordinates are relative to the field window). This is the first slot of the window if we ignore the slots that contain the border of course. To display the changes made to the cursor, we refresh our field window.
Now, we are all set.

Going back to our `while` loop, I will declare a `field` pointer that points to the current `FIELD`. This is just so that I won't have to type `g_form->fields[i]` each time I need to access that field. Then we will need a `switch` statement to test for the character `ch`.

We will need to keep track of the position of our cursor inside a given field. We will call it `x`—very creative naming, I know. We will also need to register the maximum size a string can take inside our field. The `max_size` constant will help us keep the cursor within the bounds of the field.

Listing 22: `login.c`

```c
    short x = 1; // Position of the cursor relative to the field
```

Listing 23: `login.c`

```c
    const short max_size = g_form->fields[i]->cols - 3; // Max size of the string to print
```

Listing 24: `login.c`

```c
        FIELD * field = g_form->fields[i];
        switch (ch)
        {
```

We will start with the simple up and down arrow keys. So, let's define some more macros!

Listing 25: `login.c`

```c
    #define UP_CHECK(n, boolval) \
        if (i > 0) \
            i--; \
        else \
        { \
            reset_buttons(g_form); \
            i = n - 1; \
            in_fields = boolval; \
        }
```

This is a simple macro that takes as arguments a number `n` (this will be either the number of fields or the number of buttons) and a `bool`ean value `boolval`. It basically decrements the variable `i` if it is strictly positive (in our case, this means we go back to the previous field/button). Otherwise, if `i == 0` (which means we are highlighting the first field/button), it resets the style of the buttons and removes highlights, sets the `i` to the last button if we are in a field, and sets it to last field if we are highlighting a button. For this reason, we declare a flag `in_fields`, which informs us whether we are currently inside a field (`true`) or are highlighting a button (`false`).

Listing 26: `login.c`

```
    bool in_fields = true; // A flag to check if a field is selected
```

When we switch from the first field to the last button, for example, we need to change the value of `in_fields` from `true` to `false`, etc. That is what line 49 in the macro does.

We can now add a feature to our program that allows us to just between fields and buttons using the up arrow key only, for now...

Listing 27: `login.c`

```
            case KEY_UP: // Up key
                if (in_fields)
                {
                    UP_CHECK(g_form->n_buttons, false);
                }
                else
                {
                    UP_CHECK(g_form->n_fields, true);
                }
                field = g_form->fields[i];
```

```
                x = 1;
                break;
```

Let's say, we are in the first field (`i == 0`), then we set `i` to `n_buttons - 1` to switch to the last button in the `buttons` array, then we turn off the `in_fields` flag to signal that we just got out of a field.
If we are highlighting the first button (`i == 0`), however, we set `i` to `n_fields - 1` and then raise the `in_fields` flag. Then, we would set the cursor position back to `1` (`x = 1` is the beginning of the field).

This line here

Listing 28: `login.c`

```
                field = g_form->fields[i];
```

basically updates our `field` pointer whenever we switch from field to another to point to the newly selected text `FIELD`, so that we stay up-to-date with the user's input.

Lastly, if the `in_fields` flag is raised, then we display the cursor using `curs_set(1)`, we move it to the corresponding field window to the index `i`, and refresh that window. Otherwise, we hide the cursor, then highlight the corresponding button to the index `i`.

Listing 29: `login.c`

```c
        if (in_fields)
        {
```

```c
            // Show cursor, then move it to the corresponding input field
            curs_set(1);
            wmove(field->win, 1, x);
            wrefresh(field->win);
        }
        else
        {
            // Hide cursor and highlight the corresponding button
            curs_set(0);
            highlight_b(g_form, i);
        }
```

The `highlight_b()` function is declared in the `form.h` header file and implemented in the `form.c` source file. It simply goes through the array of buttons, and turns on the `highlight` attribute for the button with index `i`.

We will do the same thing for the down arrow key and tab key. The only thing that changes here is that instead of decrementing `i`, we increment it, and when it reaches `n - 1`, we set it back to `0`. You can check out the source file if you want; I won't go through it here to avoid making this document any longer than it already is, as it's going to keep the same basic concept.

5.2 Inserting characters

Now, let's give our user the ability to type stuff in the text fields. We will set this in the `default` case in our `switch` statement, then we will filtre out unwanted keystrokes (e.g. non-printable characters). We will only give the user the ability to type inside the fields (when `in_fields == true`):

Listing 30: `login.c`

```c
            default:
                if (in_fields)
                {
                    if (isprint(ch))
                    {
```

`isprint()` is a function declared in the `<ctype.h>` header file. It takes a character as an argument, and returns a non-zero integer (`true`) if the character is printable; otherwise, it returns `0`.

Before we let the user fill in the buffer, we first need to check if there is any space left. That means we need to check if the buffer `length + 1` is greater than or equal to the buffer `capacity`. If it is, then we should grow our array. This is where our macros come in handy. As a side note, that `+ 1` is for the null terminator character `'\0'`. We basically check if the buffer is filled out completely or not.

Listing 31: `login.c`

```c
                        // Allocate more memory if necessary
                        if (field->length + 1 >= field->capacity)
                        {
                            field->buffer = GROW_ARRAY(char, field->buffer, field->capacity);
                            field->capacity = GROW_CAPACITY(field->capacity);
                        }
```

Now that we have given the user some space to type in, we can start taking their input and storing it in the buffer. Here is the code:

Listing 32: `login.c`

```
    // Simply add the character to the beginning
    field->buffer[0] = ch;
    field->buffer[1] = '\0';
```

Pretty simple, huh? We simply store the scanned character in the first slot of our buffer, then we put a `'\0'` in the end to close our string.
Now, the user can type one character. How cool is that?

To allow the user to type more characters (however much they want), we will wrap the last piece of code in an `if` statement to check whether our buffer is empty or not. This is to add the first character the user enters to the first slot in the buffer automatically if the buffer is indeed empty.

Listing 33: `login.c`

```
if (field->length++ == 0) // If the field is empty
{
    // Simply add the character to the beginning
    field->buffer[0] = ch;
    field->buffer[1] = '\0';
}
```

Otherwise, we will insert the character at the position of the cursor. For now, we will just add it to the end:

Listing 34: `login.c`

```
else
{
    // This is temporary code
    // Add the character to the end of the string
    field->buffer[field->length - 1] = ch;
    // then add a null terminator
    field->buffer[field->length] = '\0';
}
```

Note that we increment the `length` of the buffer before executing the instructions inside the `if...else` statement.
Finally, we will move the cursor one step each time the user adds a character and print the buffer content to the screen to show the user the changes made:

Listing 35: `login.c`

```
if (in_fields)
{
    // Print the new string
    mvwprintw(field->win, 1, 1, "\%s", field->buffer);
```

With that, we now can give the user the privilege of navigating through the field using left and right arrows keys. It is going to be a simple `if` statement to check if we have reached the end/beginning, so as to stop the cursor from going out of bounds.

Listing 36: `login.c`

```
case KEY_LEFT: // Left key
    if (in_fields)
        if (x > 1)
            x--;
    break;
case KEY_RIGHT: // Right key
    if (in_fields)
        if (x < field->length + 1 && x <= max_size)
            x++;
    break;
```

Here is a simple visualization of what the program does:

```
This is just a example
```

cursor

`x = 14`
`length = 22`
`capacity = 32`

Let's say the cursor is currently at `x = 14`. If the user hits `RIGHT ARROW`, `x` gets incremented, and when the screen refreshes, this is what is shown:

```
This is just a example
```

cursor

`x = 15`
`length = 22`
`capacity = 32`

If the cursor is at `x = 23` (the cursor is allowed to go one step only after the text), however, and the user tries to move it to the right, `x` will not get incremented. Therefore, the cursor will stay in place.

```
This is just a example
```

cursor

`x = 22`
`length = 22`
`capacity = 32`

Similarly, if the cursor is at `x = 1` and the user hits `LEFT ARROW`, the cursor won't move. But there is still one problem (probably more). If the cursor is at, say, `x = 15`,

```
This is just a example
```

cursor

`x = 15`
`length = 22`
`capacity = 32`

and the user tried to type a character, the character will be added to the end of the line:

```
This is just a examplen
```

cursor

`x = 16`
`length = 23`
`capacity = 32`

To fix this, we will need to insert the character at the position of the cursor (`- 1` to convert it to an array index):

Listing 37: `login.c`

```c
                        else
                        {
                            // Move the string from position (x - 1) one character to
                                the right
                            //      to make room for the new character
                            memmove(&field->buffer[x], &field->buffer[x - 1], field->
                                length - (x - 1) + 1);
                            // Insert the new character
                            field->buffer[x - 1] = ch;
                        }
```

I used the `memmove()` function from `<string.h>`. This function takes three arguments: the destination, the source and an integer `n`. It copies `n` characters from the source string to the destination string.
Why didn't I use `strcpy()`, you ask? Well, generally, `strcpy()` is much faster than `memmove()`. However, using `strcpy()` on a string that acts both as a source and a destination can lead to strange and undefined behaviour, and we don't want any of that. I won't go into the details of why that happens, but you can always look it up on the internet.
Back to our code:

`length - (x - 1) + 1 = 9` characters

```
This is just a example¬
```

cursor

`x = 15`
`length = 22`
`capacity = 32`

Here, ¬ denotes `'\0'` just for the sake of clarity.

When the user types a character, we first move the characters from the cursor to the end one step to the right. That means, we move `length - (x - 1)` characters `+ 1` for the extra `'\0'`.

→

```
This is just a  example¬
```

cursor

`x = 15`
`length = 23`
`capacity = 32`

```
This is just an example¬
```

cursor

`x = 16`
`length = 23`
`capacity = 32`

Let's also limit the maximum the cursor can go:

Listing 38: `login.c`

```
                    if (x <= max_size)
                        x++;
```

There is still another issue here: when the cursor reaches the end of the field, it stops. Consequently, the user won't be able to add more characters at the end of the string; instead, they will be inserted at the location of the cursor.
To solve this problem, I am going to introduce two more variables:

Listing 39: `login.c`

```
    int b_pos = 0, // Position of the cursor relative to the buffer
        strstart = 0; // Index of the beginning of the string to print
```

`b_pos` is the position of the cursor, except it's the relative position in the buffer. I set the initial value as `0` just for the sake of convience. `strstart` is the index of the first character in the buffer that we want to print. This will help us "fake" the cursor moving through the field.

I believe visualizing is always a better way to explain things. Let's consider the following example:

```
This is a very long string to demonstr█
```

cursor

`x = 39`
`length = 61`
`capacity = 64`

```
buffer := "This is a very long string to demonstrate how the code works."
```

Following the figure above, here is a visualization of the buffer:

`strstart` `b_pos`

```
This is a very long string to demonstrate how the code works.¬
```

When the user moves the cursor to the left, we should increment `x`, `b_pos` and `strstart` altogether:

`strstart` `b_pos`

```
This is a very long string to demonstrate how the code works.¬
```

This is what is going to be shown to the user on the screen:

```
his is a very long string to demonstra█
```

cursor

`x = 39`
`length = 61`
`capacity = 64`

As you can see, the cursor did not move from its place (because `x` is independent from the buffer) and `b_pos` took its role. We (figuratively) moved the whole buffer to the left to reveal the rest of the string. Although, we didn't really move it; we just printed the portion that start from `strstart`.

So, the next thing we need to do here is update the code to allow the user to navigate through a long string. We'll start with the left arrow key :

Listing 40: `login.c`

```c
            case KEY_LEFT: // Left key
                if (in_fields)
                {
                    // Order of these 'if' statements matter A LOT
                    //      I learned that the hard way... *sigh*
                    if (field->length > max_size - 1 && x == 1 && strstart > 0)
                        strstart--;
                    if (x > 1)
                        x--;
                    if (b_pos > 0)
                        b_pos--;
                }
                break;
```

There are three `if` statements here each of which serves a certain role. **Keep in mind** that, the order of these `if` statements *really* makes a difference.
Also, ignore that comment up there. It's not like I spent more than four hours debugging that bug... (*sarcasm*)

First, we check to see if the length of the string exceeds the size of the field, if the cursor is at the beginning, and if `strstart` is positive (we can't decrement it otherwise because it

can't be negative). If all of these conditions are met, it means that the user wants to—and can—go back. So, we decrement `strstart`. We also need to check if the cursor is not at the beginning, so that we decrement `x`.
Last but not least, we need to decrement `b_pos` too. `b_pos`—unlike `x`—gets decremented each time we go to left.
We can do something similar for the right arrow key:

Listing 41: `login.c`

```c
            case KEY_RIGHT: // Right key
                if (in_fields)
                {
                    // Order...
                    if (field->length > max_size - 1 && x == max_size + 1 && b_pos < field->length)
                        strstart++;
                    if (x < field->length + 1 && x <= max_size)
                        x++;
                    if (b_pos < field->length)
                        b_pos++;
                }
                break;
```

We increment `strstart` if the string is too long for the field, if the cursor is at the end of the field, and if the position of the "relative cursor" is not at the end of the string. Otherwise we increment `x` if the cursor is not at the end of the string (the string can be smaller than the field) and if it is not at the end of the field. And we increment `b_pos` when it is less than the length of the string.

Are we done yet? Nope. Not at all. We still have an issue when inserting a character. When the user enters a character, we insert it at position `x - 1`, which is not what we want. We have another cursor—the "relative cursor"—that should take care of that. Changing that isn't difficult, if at all; we just replace `x - 1` by `b_pos` and `x` by `b_pos + 1`:

Listing 42: `login.c`

```c
                        else
                        {
                            // Move the string from position (b_pos) one character to the right
                            //      to make room for the new character
                            memmove(&field->buffer[b_pos + 1], &field->buffer[b_pos], field->length - b_pos + 1);
                            // Insert the new character
                            field->buffer[b_pos] = ch;
                        }
```

We also need to increment `strstart` when we insert a character. This happens only when the string is larger than the field and when the cursor is at the end. In addition to that, `b_pos` needs to be incremented. There are no conditions necessary for this.

Listing 43: `login.c`

```c
                        // Order...
                        if (field->length > max_size && x == max_size + 1)
                            strstart++;
                        if (x <= max_size)
                            x++;

                        b_pos++;
```

Let's not forget to update our `mvwprintw()` to print only a portion of the big string. Right now, it just prints the whole thing all the time and goes out of bounds.

Here is an updated version :

Listing 44: `login.c`

```c
                // Print the new string
                mvwprintw(field->win, 1, 1, "%.*s", max_size, &field->buffer[strstart]);
```

Instead of printing `field->buffer`, we print the substring that starts from `strstart`. That is basically `&field->buffer[strstart]`. Because `mvwprintw()` —just like `printf()` — takes just the address of the first character in a string. This means that `field->buffer` and `&field->buffer[0]` are equivalent. So, rather than starting from index `0`, we start from `strstart`. Simple, right? Except for the weird `"%.*s"` format.

In C, you can print a floating-point number like so:

```c
printf("%f", 3.1418); // This prints 3.141800
```

But, we can also specify the precision using the format `"%.nf"`, where n is an integer (that n is **not** a variable name).

```c
printf("%.2f", 3.1418); // This prints 3.14
```

What you may or may not know is that we can actually pass the precision to printf as an argument too, so that we can use a variable that decides it for us. We can do that using the format specifier `"%.*f"`.

```c
printf("%.*f", 2, 3.1418); // This also prints 3.14
```

This also applies to strings. We can control the width of the string, which is the maximum number of characters to be printed. For example:

```c
const char * str = "This is an example";
printf("%.*s", 7, str);
// Output :
//       This is
```

Knowing that `mvwprintw()` works the same as `printf`, we can use the same format to limit the number of characters to print on the screen so as not to go outside the field. That is exactly what you see up there.

We can also make use of the `hidden` attribute that we have given each field and hide the password field content. For this, I created a function `hide_str()` that takes the size of the string as an argument and returns a string that contains asterisks (`*`) only:

Listing 45: `login.c`

```c
static char * hide_str(int length)
{
    char * hidden = (char *)malloc((length + 1) * sizeof(char));

    for (int i = 0; i <= length; i++)
        hidden[i] = '*';
    hidden[length] = '\0';

    return hidden;
}
```

I know that this slows down the program significantly, but for now, performance isn't an issue. To use this with our `mvwprintw()` function, we just need to check if the `field->hidden` flag is raised. If it is, then we print the hidden string; otherwise, we just print the normal string.

As a side note, we *can* optimize the use of the `hide_str` function so that we only pass `max_size` as the size of the hidden string if the size exceeds `max_size`. I didn't do this here to keep things simple.

Here is the new version of `mvwprintw()`:

Listing 46: `login.c`

```
            // Print the new string
            mvwprintw(field->win, 1, 1, "%.*s", max_size, (field->hidden ? hide_str(
                field->length) : &field->buffer[strstart]));
```

Here, instead of an `if` statement, I used something called a **ternary operator**. The syntax is simple:

Listing 47: `login.c`

```
(condition) ? (expression1) : (expression2)
```

The ternary operator takes *three* operands, and it is used to shorten an `if...else` statement. When the `condition` is truthy (either `true` or non-zero), then the operator evaluates to `expression1`; otherwise, it evaluates to `expression2`.

In our case, we are checking if the `hidden` flag is raised for a specific field to hide the content. If that's not the case, then we will directly print the string.
I know I just made it harder to read, but I just didn't feel like using an `if` statement for that, and it is also an occasion for you to learn about ternary operators. They are pretty cool.

One last thing left to do is to update the up and down arrow key `case`s for better accessibility:

Listing 48: `login.c`

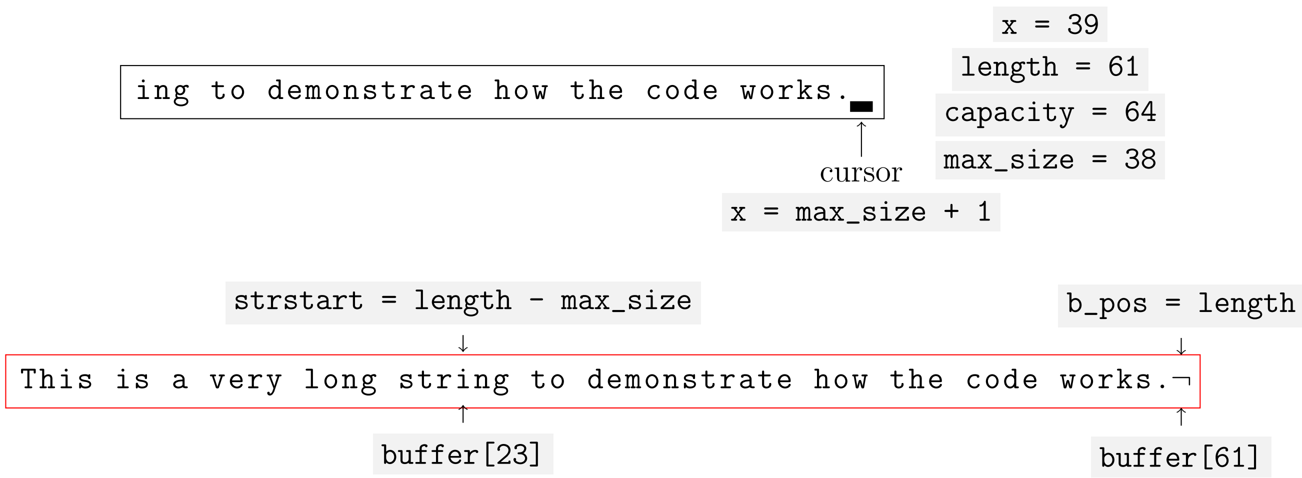
```
            case KEY_UP: // Up key
                if (in_fields)
                {
                    UP_CHECK(g_form->n_buttons, false);
                }
                else
                {
                    UP_CHECK(g_form->n_fields, true);
                }
                field = g_form->fields[i];
                x = ((field->length + 1) > max_size) ? (max_size + 1) : (field->
                    length + 1);
                b_pos = field->length;
                strstart = ((field->length + 1) > (max_size)) ? (field->length -
                    max_size) : 0;
                break;
```

When we switch from field to another, instead to positioning the cursor back to the beginning (`x = 1`), we want to put it at the end of the text inside the field.
Ternary operators again?! Yes. I like them, and I'll never stop using them despite the lack of readability.

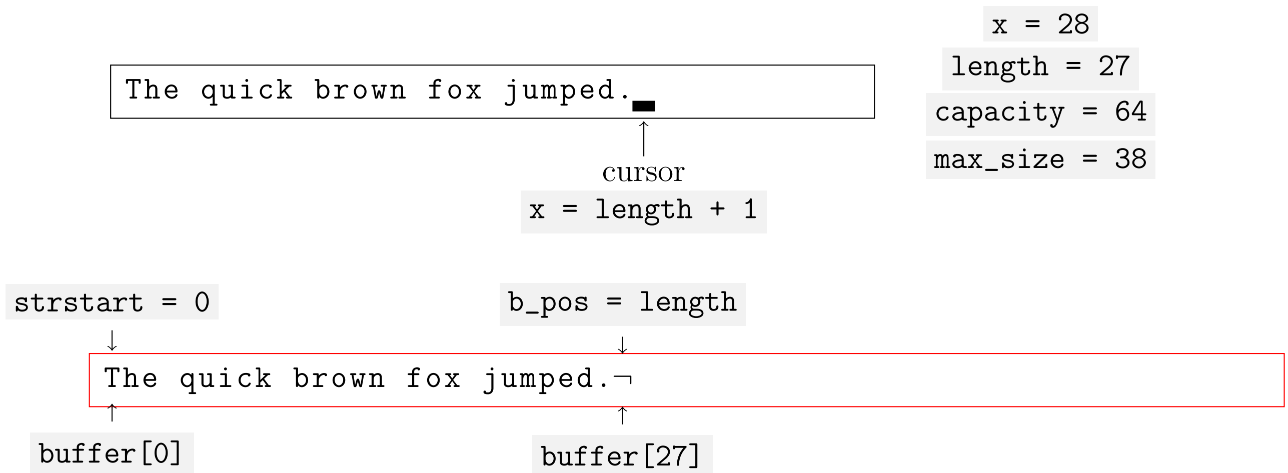
If the string is larger than the size of the field, then we will set `x` to `max_size + 1` and `strstart` to `field->length - max_size`. Otherwise, we set `x` to `field->length + 1` and `strstart` to `0`. `b_pos` is always going to be assigned `field->length`.
This is going to be the same for the down arrow key.

Don't worry about all those values; I have prepared a visualization for you:

– Case 1: `length + 1 > max_size`



– Case 2: `length + 1 <= max_size`



5.3 Deleting characters

What kind of form is it if it doesn't allow you to correct your mistakes? This is what we will be doing in this part. This shouldn't be that hard, since we will be using whatever tools we have left from what we were doing before.
Since a backspace is not a printable character, then we will put it in an `else` statement in the `default` case.

Listing 49: `login.c`

```
                    // If the user hits a backspace
                    else if ((ch == KEY_BACKSPACE || ch == '\b' || ch == 127) &&
                        field->length > 0 && b_pos > 0)
                    {
```

We check if the key the user hit is a backspace. And because the code for a backspace differs from OS to another, we will be checking them all.
A backspace could return either `KEY_BACKSPACE`, `'\b'`, or `127`. We also have to check if the length of the string is not `0` (`field->length > 0`) and the relative position of the cursor is not at the beginning (`b_pos > 0`). One of the last two conditions *could* be removed, but I don't want to.

The first and foremost thing we want to do inside this `if` statement is remove the character at `b_pos`. We will need our old friend `memmove()` again:

Listing 50: `login.c`

```
                        // Move the string from position (x) one character to the
                            left
                        memmove(&field->buffer[b_pos - 1], &field->buffer[b_pos],
                            field->length - b_pos + 1);
```

We move all characters starting from `b_pos` to the end (including `'\0'`) back to overwrite the character that we want to delete.
Of course, we shouldn't forget to decrement the `length` and the cursor position:

Listing 51: `login.c`

```
                        field->length--;
                        b_pos--;
```

What about `x` and `strstart`? Simply put, we will decrement `strstart` if and only if the string is larger than the field, and if `strstart > 0`. Otherwise, we will decrement `strstart`. Now, when we remove a character and print the new string, we will be left with a duplicate character at the end because the new string is shorter. For this reason, we will print a space at the end to erase that:

Listing 52: `login.c`

```
                        if (field->length > max_size && strstart > 0)
                        {
                            // Move the whole string to the right
                            strstart--;
                            // Remove last (duplicate) character from the window
                            mvwaddch(field->win, 1, max_size, ' ');
                        }
                        else
                        {
                            // Move the cursor back one character
                            x--;
                            // Remove last (duplicate) character from the window
                            if (field->length <= max_size)
                                mvwaddch(field->win, 1, field->length, ' ');
                        }
```

That's it! That is all there is to this.

5.4 Switching forms

We've successfully managed to get the user input and store it, but we still didn't give the user the ability to switch to the other form.
This is going to be very simple; all we need to do is to check if the user his `ENTER`, and then check which button they are highlighting. If they selected the `"Register"` button for example, we just remove the Login Form, create a new Registration Form, and then reinitialize all the variables. If they selected the `"Submit"` button, or if the cursor is in one of the fields, then we submit the form to another function, which we still haven't created yet.

Listing 53: `login.c`

```
            // If the user hits Enter while typing
            //      or if they hit Enter while highlighting the Submit button
            if (in_fields || (!in_fields && i == 0)); // Submit data
            else
            {
                // Destroy the current form
                destroy_form(g_form);
                // Create a new one accordingly
```

```
            g_form = is_login ? create_registrform() : create_loginform();
            // Highlight the first text field and set the cursor to the beginning
            in_fields = true; i = 0; x = 1; b_pos = 0;
            curs_set(1);
            wmove(g_form->fields[0]->win, 1, 1);
            wrefresh(g_form->fields[0]->win);
            // Switch between login and register page
            is_login = !is_login;

            continue;
        }
```

This concludes this section. Later on, we will talk about submitting the form. After we implement that of course.
I don't know why I spent this much time writing this document... I am starting to question my life choices. But, hopefully, it will be of use to someone out there. Hopefully...