A Short Introduction to Curses

Keith Gaughan

March 22, 2003

Contents

1 A wee problem and a history lesson **1**

1.1 Long, long ago, on a college campus far, far away. . . 1

1.2 . . . which begat Curses 1

2 Starting with Curses **1**

2.1 Builing a program 1

2.2 A short example 2

2.2.1 Breakdown of the example 2

2.2.2 The mysterious refresh() 2

3 Dealing with the screen and positioning **3**

3.1 How the screen is laid out 3

3.2 Writing to the screen 3

3.2.1 addch() 3

3.2.2 addstr() 4

3.2.3 addnstr() 4

3.3 Positioning the cursor 4

3.4 Clearing the screen 4

4 Handling the keyboard **4**

4.1 Keyboard modes 4

4.1.1 Echoing characters to the screen 4

4.1.2 Turning line buffering on and off 4

4.2 Getting input 5

4.2.1 getch() 5

4.2.2 getstr() 5

4.2.3 getnstr() 5

5 Odds and ends **5**

5.1 beep() 5

5.2 flushinp() 5

6 Elvis has left the building. . . **5**

1 A wee problem and a history lesson

Have you ever felt that `printf()` et al, while terribly useful in their own right, are annoyingly deficient in others. I mean, you can't use them to print at any arbitrary location on the screen, neither is there any nice, simple way of clearing the screen. Getting stuff from the keyboard is yet more frustrating: `gets()` isn't safe and so the only way of safely reading a string in from the keyboard is with `fgets()`, which can feel like overkill at the best of times. `scanf()` is only really good for reading highly formatted data and is useless when it comes to reading arbitrary lines of text. All in all, writing a UI for your console-based programs is a right royal pain in the ass.

1.1 Long, long ago, on a college campus far, far away...

That, amongst others, is why *Curses* was written. Back when Bill Joy, was writing the first version of *vi*[1] at Berkeley, he had a problem—there were no standard set of control codes for controlling things like the cursor position, clearing the screen, setting the text colour, ringing a bell, etc. Each type of terminal has their own set. This, as you can imagine, made writing programs that did anything more than print lines and read from from the keyboard rather difficult.

So Joy, being the good programmer that he was, abstracted[2] the problem. He created a library that read *terminal capabilities* from a database file. This way he never had to worry about what terminal was being used—if the data wasn't there, it was just a matter of adding it to the database[3].

1.2 ...which begat Curses

Joy's library was a good solution and came to be widely used in other programs. Now nobody had to worry about what commands to send to the terminal, they just had to read them from the database. But there's always a better way of doing things, I mean, why should you have to read the control codes out of the database to position the cursor at a given place on the screen when you could wrap all that up in a function that does it for you? Hence Curses.

2 Starting with Curses

Curses is a library that provides a rather simple and universal, if idiosyncratic, way of creating halfway-decent interfaces for console-based programs. It hides nearly all of the nasty details of dealing with character-based UIs. The most prevalent version of Curses floating about these days is *NCurses*, which provides some pretty nifty UI management functionality, but that's outside the realm of this tutorial. There's also a version for Windows called *PDCurses*.

2.1 Builing a program

Any program using Curses must include the `curses.h` header file and, when being compiled, linked to the Curses library by including the `-lcurses` flag, e.g.

```
gcc foo.c -o foo -lcurses
```

Before your program can start using any of Curses' functionality, it must call `initscr()`. This allows Curses to setup its data-structures and allocate any other resources it needs. When your program's terminating, you need to call `endwin()`, which will give Curses a chance to clean up after itself.

---

[1] Which was considered an *extremely* user-friendly text editor when it first appeared. If you want to see what people were enduring up until then, try `ed` and you'll never curse `vi` ever again...

[2] And this is always a good thing as one of the primary virtues of a programmer is *laziness*—good programmers will put in a little bit of extra work in the short term because they know that doing so will save them a heap of work in the long term. This is why they create libraries, keep their code well-factored to minimise redundancy and maximise reusability, and so on.

[3] Actually, he did this twice—the first version was called *termcap* and the next, *terminfo*. Terminfo was created to solve scalability and organisational problems in the Termcap database as well as other problems with the library.

2.2 A short example

Here's a program that does nothing more than print a message out to the screen, wait a couple of seconds, and then exit. If you can grasp what's happening here, you've grasped a lot about Curses.

```
/* A basic Curses demo */

#include <stdio.h>
#include <unistd.h>
#include <curses.h>

void main(void)
{
    /* Attempt to initialise Curses. */
    if (initscr() == NULL)
    {
        /* Feck! Something went wrong... */
        perror("Couldn't initialise Curses!");
        exit(EXIT_FAILURE);
    }

    /* Print something out. */
    printw("I am Curses. Hear me squeak!");

    /* Make sure any changes are shown. */
    refresh();

    /* Wait for a couple of seconds... */
    sleep(2);

    /* Clean up! */
    endwin();
}
```

2.2.1 Breakdown of the example

The first thing going on is the program attempts to initialise Curses with a call to `initscr()`. This may or may not fail, but you can tell when it does because it'll return `NULL`. If this happens, we need to handle it. Here we just complain and exit with an error.

Next we encounter the `printw()` function. This acts exactly like `printf()`, except it's part of Curses. Nothing odd here.

The next line may strike you as odd though. Curses doesn't write stuff out to the screen immediately—you have to tell it when to update the screen. This is what `refresh()` is for, but more about that in a bit.

`sleep()` you've probably encountered before. It puts the process to sleep for a number of seconds.

And finally, `endwin()` cleans up after Curses and restores the contents of the screen from before `initscr()` was called.

2.2.2 The mysterious refresh()

`refresh()` a legacy of when Curses was first created. Back then, the connection between a dumb terminal and the server was *slooow*[4].

[4]To put things in perspective, a typical modem these days runs at about 56 kilobauds, whereas the connections between dumb

To keep refreshes efficient, Curses keeps two buffers—one, called `curscr`, which reflects the true contents of screen, and another, called `stdscr`, which holds a copy of the screen as you'd like it to look. By analysing the differences between these two, Curses can efficiently update the contents of the screen when you call `refresh()`. Even these days, this makes good sense because there's no sense in updating the screen until you really need to, so `refresh()` isn't completely redundant.

3 Dealing with the screen and positioning

3.1 How the screen is laid out

The screen, as far as Curses is concerned, is a grid of character cells with its origin in the top-left of the screen.

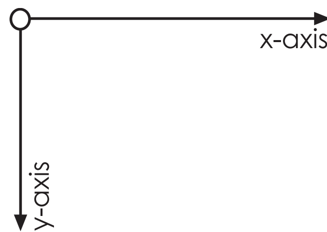

Figure 1: How Curses views the screen

The origin in the top-left is denoted by (0,0). The coördinate values for each axis increase as you move down and to the right[5]. One of Curses' idiosyncracies is that coördinates are given as (y,x), rather than the more conventional (x,y) order[6]. If it helps think of the coördinates as being given in (line, column) order.

To figure out how big the screen is, you use the, yet again, rather idiosyncratic `getmaxyx()`, e.g.

```
int maxy, maxx;

getmaxyx(stdscr, maxy, maxx);
printw("The maximum x and y co-ordinates are %d and %d.\n",
       maxx, maxy);
```

The way `maxy` and `maxx` are passed in isn't a typo—`getmaxyx` is a macro that extracts information from the screen structure (in this case `stdscr`), and puts it in the variables passed in.

You can use the result to make sure the screen is big enough to fit the needs of your program.

3.2 Writing to the screen

In addition to `printw()`, there are a number of other functions for putting stuff out on the screen.

3.2.1 addch()

Writes a character to the screen.

```
int addch(char ch);
```

terminals and servers back then were only just a little faster than a speeding turtle—between 2 kilobauds and, if you were *really* lucky, 9 kilobauds.

[5] must. . . resist. . . making. . . JFK joke. . .

[6] Another one of those *Jamaican Gold* moments, I'll warrant!

3.2.2 addstr()

Writes a string to the screen. This is useful if you don't need `printw()`'s formatting abilities.

```
int addstr(char* str);
```

3.2.3 addnstr()

Like `addstr()`, but only writes a given number of characters out. This can be a bit safer in places.

```
int addnstr(char* str, int n);
```

3.3 Positioning the cursor

Being able to print to the screen is great, but being able to position the cursor where you want would be pretty useful too, hence `move()`, which lets you do this.

```
int move(int y, int x);
```

In addition, all the output routines have special forms starting with `mv` that allow you to include cursor coördinates in them. In each one, the coördinates you want are specified in the first two arguments, for instance the `mv` version of `addch()` would be:

```
int mvaddch(int y, int x, char ch);
```

And all the others (`mvprintw()`, `mvaddstr()`, `mvaddnstr()`) work in exactly the same way.

3.4 Clearing the screen

The `clear()` function lets you clear the screen. `clrtobot()` erases everything from wherever the cursor is to the bottom of the screen. `clrtoeol()` clears everything from the cursor to the end of the current line. Dead handy these!

```
int clear(void);
int clrtobot(void);
int clrtoeol(void);
```

4 Handling the keyboard

Curses includes its own suite of functions for dealing with keyboard input. You should use these in preference to the regular ones in `stdio.h` when you want to get input from the user.

4.1 Keyboard modes

4.1.1 Echoing characters to the screen

Normally, characters are *echoed* when typed, but sometimes you don't want this to happen, e.g. the user is typing a password. To turn this off, call `noecho()`, and to turn it back on use `echo()`.

4.1.2 Turning line buffering on and off

Characters are buffered by the terminal until a newline is typed by default. To turn buffering off, call `cbreak()`, and to turn it on again call `nocbreak()`.

4.2 Getting input

4.2.1 getch()

Reads a single character from the keyboard. It also has a `mv` form that moves the cursor before the read.

```
int getch(void);
int mvgetch(int y, int x);
```

4.2.2 getstr()

Reads a string from the keyboard. This is equivalent to a series of calls to `getch()` until a newline is typed. The resulting string is then placed in the buffer pointed to by `str`. Just like `getch()`, this has a `mv` variant.

```
int getstr(char* str);
int mvgetstr(int y, int x, char* str);
```

4.2.3 getnstr()

This is a *safe* version of `getstr()` that allows you to limit the number of characters that can be typed to `n`, hence preventing buffer overflow and bad breath.

```
int getnstr(char* str, int n);
int mvgetnstr(int y, int x, char* str, int n);
```

5 Odds and ends

5.1 beep()

Makes the terminal beep. Exciting, huh?

```
int beep(void);
```

5.2 flushinp()

Throws away any characters the user might have typed that haven't been read.

```
int flushinp(void);
```

6 Elvis has left the building...

There's much more stuff in Curses than just that, but for simple programs, that's all you need. Other topics I'll probably be covering in further articles include *forms*, *pads*, *windows*, *colour* and *using the mouse*.

If you've any questions or comments, you'll always be able to get me at <kmgaughan@eircom.net>, though <kgaughan@mcom.cit.ie> should work too. The latest version of this document should be downloadable from either my website or my weblog.