Introduction to the Unix Curses Library

Norman Matloff
Department of Computer Science
University of California at Davis

April 8, 2011

Contents

1 History

1.1 Purpose of the Curses Library

Many widely-used programs need to make use of a terminal's cursor-movement capabilities. A familiar example is the **vi** (or the **vim** variation) text editor; most of its commands make use of such capabilities. For example, hitting the **j** key while in **vi** will make the cursor move up one line. Typing **dd** will result in the current line being erased, the lines below it moving up one line each, and the lines above it remaining unchanged.

A potential problem with all this is that different terminals have different ways in which to specify a given type of cursor motion. For example, if a program wants to make the cursor move up one line on a VT100 terminal, the program needs to send the characters Escape, [, and A:

```
printf("%c%c%c",27,'[','A');
```

(the ASCII character code for the Escape key is 27). But for a Televideo 920C terminal, the program would have to send the ctrl-K character, which has code 11:

```
printf("%c",11);
```

Clearly, the authors of programs like **vi** would go crazy trying to write different versions for every terminal, and worse yet, anyone else writing a program which needed cursor movement would have to "re-invent the wheel," i.e. do the same work that the **vi**-writers did, a big waste of time.

That is why the `curses` library was developed. The goal was to alleviate authors of cursor-oriented programs like **vi** of the need to write different code for different terminals. The programs would make calls to the library, and the library would sort out what to do for the given terminal type.

For example, if your program needs to clear the screen, it would not (directly) use any character sequences like those above. Instead, it would simply make the call

```
clear();
```

and `curses` would do the work on the program's behalf, i.e. would print the characters Escape, [, and A, causing the screen to clear.

1.2 Evolution of the Role of Curses

Development of the `curses` library was a major step in the evolution of Unix software. Even though the VT100 terminal type eventually became standard, `curses` continued to play an important role, by providing an extra level of abstraction to the programmer, alleviating the programmer from needing to know even the VT100 cursor movement codes.

The library is still quite important in today's GUI-oriented world, because in many cases it is more convenient to use the keyboard for actions than the mouse.[1] Today physical terminals are rarely used, but typical

[1] After all, even many Microsoft Windows applications provide "keyboard shortcuts."

Unix work does involve some text windows each of which is emulating a VT100 terminal.[2] The **vi** editor and other `curses`-based programs continue to be popular.[3]

2 Include and Library Files

In order to use `curses`, you must include in your source code a statement

```
#include <curses.h>
```

and you must link in the `curses` library:

```
gcc -g sourcefile.c -lcurses
```

3 Two Examples

It's best to learn by example!

Try running these programs! You don't have to key in their source code; instead, you can get it from the raw file which produced this document, `http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Curses.tex`, and then editing out the non-code.

If you are in a hurry, you may wish to go directly to the second example, which anyway is better commented and makes use of more `curses` features.

3.1 Simple, Quick Introductory Example

Our first example does very little. You can think of it as a very primitive, uninteresting game. But it gives us a start.

The comments at the top of the source file tell you what the "game" does. Compile and run the program, inputting various characters as you like. Then read the code (start with **main()**, as you should in general),

[2] The program running the window is **xterm** or something similar, e.g. **gnome-terminal**.

[3] Interestingly, even some of those classical Curses programs have also become somewhat GUI-ish. For instance **vim**, the most popular version of **vi** (it's the version which comes with most Linux distributions, for example), can be run in **gvim** mode. There, in addition to having the standard keyboard-based operations, one can also use the mouse. One can move the cursor to another location by clicking the mouse at that point; one can use the mouse to select blocks of text for deletion or movement; etc. There are icons at the top of the editing window, for operations like Find, Make, etc.

with the comments doing the explaining.

```
// simple curses example; keeps drawing the inputted characters, in columns
// downward, shifting rightward when the last row is reached, and
// wrapping around when the rightmost column is reached

#include <curses.h>  // required

int r,c,  // current row and column (upper-left is (0,0))
    nrows,  // number of rows in window
    ncols;  // number of columns in window

void draw(char dc)

{  move(r,c);  // curses call to move cursor to row r, column c
   delch();  insch(dc);  // curses calls to replace character under cursor by dc
   refresh();  // curses call to update screen
   r++;  // go to next row
   // check for need to shift right or wrap around
   if (r == nrows)  {
      r = 0;
      c++;
      if (c == ncols) c = 0;
   }
}

main()

{  int i;  char d;
   WINDOW *wnd;

   wnd = initscr();  // curses call to initialize window
   cbreak();  // curses call to set no waiting for Enter key
   noecho();  // curses call to set no echoing
   getmaxyx(wnd,nrows,ncols);  // curses call to find size of window
   clear();  // curses call to clear screen, send cursor to position (0,0)
   refresh();  // curses call to implement all changes since last refresh

   r = 0; c = 0;
   while (1)  {
      d = getch();  // curses call to input from keyboard
      if (d == 'q') break;  // quit?
      draw(d);  // draw the character
   }

   endwin();  // curses call to restore the original window and leave

}

```

3.2 A Second, More Useful Example

Here's a program you can actually use. If you need to kill a number of processes, this program will allow you to browse through your processes, killing the ones you want.

Again, compile and run the program, and kill some garbage processes which you've set up for that purpose. (For that matter, you could kill **psax** itself.) Then read the code, with the comments doing the explaining.

```
// psax.c; illustration of curses library

// runs the shell command 'ps ax' and dislays the last lines of its output,
// as many as the window will fit; allows the user to move up and down
// within the window, with the option to kill whichever process is
// currently highlighted

```

```
8  // usage:  psax
9
10 // user commands:
11
12 //    'u':  move highlight up a line
13 //    'd':  move highlight down a line
14 //    'k':  kill process in currently highlighted line
15 //    'r':  re-run 'ps ax' for update
16 //    'q':  quit
17
18 // possible extensions:  allowing scrolling, so that the user could go
19 // through all the 'ps ax' output, not just the last lines; allow
20 // wraparound for long lines; ask user to confirm before killing a
21 // process
22
23 #define MAXROW 1000
24 #define MAXCOL 500
25
26 #include <curses.h>  // required
27
28 WINDOW *scrn; // will point to curses window object
29
30 char cmdoutlines[MAXROW][MAXCOL];  // output of 'ps ax' (better to use
31                                    // malloc())
32 int ncmdlines,  // number of rows in cmdoutlines
33     nwinlines,  // number of rows our "ps ax" output occupies in the
34                 //  xterm (or equiv.) window
35     winrow,  // current row position in screen
36     cmdstartrow,  // index of first row in cmdoutlines to be displayed
37     cmdlastrow;  // index of last row in cmdoutlines to be displayed
38
39 // rewrites the line at winrow in bold font
40 highlight()
41 {  int clinenum;
42    attron(A_BOLD);  // this curses library call says that whatever we
43                     // write from now on (until we say otherwise)
44                     // will be in bold font
45    // we'll need to rewrite the cmdoutlines line currently displayed
46    // at line winrow in the screen, so as to get the bold font
47    clinenum = cmdstartrow + winrow;
48    mvaddstr(winrow,0,cmdoutlines[clinenum]);
49    attroff(A_BOLD);  // OK, leave bold mode
50    refresh();  // make the change appear on the screen
51 }
52
53 // runs "ps ax" and stores the output in cmdoutlines
54 runpsax()
55 {  FILE *p; char ln[MAXCOL]; int row,tmp;
56    p = popen("ps ax","r");  // open Unix pipe (enables one program to read
57                             // output of another as if it were a file)
58    for (row = 0; row < MAXROW; row++)  {
59       tmp = fgets(ln,MAXCOL,p);  // read one line from the pipe
60       if (tmp == NULL) break;  // if end of pipe, break
61       // don't want stored line to exceed width of screen, which the
62       // curses library provides to us in the variable COLS, so truncate
63       // to at most COLS characters
64       strncpy(cmdoutlines[row],ln,COLS);
65       // remove EOL character
66       cmdoutlines[row][MAXCOL-1] = 0;
67    }
68    ncmdlines = row;
69    close(p);  // close pipe
70 }
71
72 // displays last part of command output (as much as fits in screen)
73 showlastpart()
74 {  int row;
75    clear();  // curses clear-screen call
76    // prepare to paint the (last part of the) 'ps ax' output on the screen
77    // two cases, depending on whether there is more output than screen rows;
78    // first, the case in which the entire output fits in one screen:
```

```
79     if (ncmdlines <= LINES)  { // LINES is an int maintained by the curses
80                                // library, equal to the number of lines in
81                                // the screen
82        cmdstartrow = 0;
83        nwinlines = ncmdlines;
84     }
85     else  { // now the case in which the output is bigger than one screen
86        cmdstartrow = ncmdlines - LINES;
87        nwinlines = LINES;
88     }
89     cmdlastrow = cmdstartrow + nwinlines - 1;
90     // now paint the rows to the screen
91     for (row = cmdstartrow, winrow = 0; row <= cmdlastrow; row++,winrow++)
92        mvaddstr(winrow,0,cmdoutlines[row]);  // curses call to move to the
93                                              // specified position and
94                                              // paint a string there
95     refresh();  // now make the changes actually appear on the screen,
96                 // using this call to the curses library
97     // highlight the last line
98     winrow--;
99     highlight();
100 }
101
102 // moves up/down one line
103 updown(int inc)
104 {  int tmp = winrow + inc;
105    // ignore attempts to go off the edge of the screen
106    if (tmp >= 0 && tmp < LINES)  {
107       // rewrite the current line before moving; since our current font
108       // is non-BOLD (actually A_NORMAL), the effect is to unhighlight
109       // this line
110       mvaddstr(winrow,0,cmdoutlines[cmdstartrow+winrow]);
111       // highlight the line we're moving to
112       winrow = tmp;
113       highlight();
114    }
115 }
116
117 // run/re-run "ps ax"
118 rerun()
119 {  runpsax();
120    showlastpart();
121 }
122
123 // kills the highlighted process
124 prockill()
125 {  char *pid;
126    // strtok() is from C library; see man page
127    pid = strtok(cmdoutlines[cmdstartrow+winrow]," ");
128    kill(atoi(pid),9);  // this is a Unix system call to send signal 9,
129                        // the kill signal, to the given process
130    rerun();
131 }
132
133 main()
134 {  char c;
135    // window setup, next 3 lines are curses library calls, a standard
136    // initializing sequence for curses programs
137    scrn = initscr();
138    noecho();  // don't echo keystrokes
139    cbreak();  // keyboard input valid immediately, not after hit Enter
140    // run 'ps ax' and process the output
141    runpsax();
142    // display in the window
143    showlastpart();
144    // user command loop
145    while (1)  {
146       // get user command
147       c = getch();
148       if (c == 'u') updown(-1);
149       else if (c == 'd') updown(1);
```

```
150         else if (c == 'r') rerun();
151         else if (c == 'k') prockill();
152         else break;  // quit
153      }
154      // restore original settings and leave
155      endwin();
156   }
```

3.3 A Note on "Cooked" and "Raw" Modes in the Examples

In most programs you write, keyboard actions are in **cooked** mode. This means that your keyboard input is not sent to the program (e.g. not sent to the **scanf()** function or to **cin**) until you hit the Enter key. This gives you a chance to use the Backspace key to erase a character which you typed but now wish to "rescind." Again, this means that your program never sees the character you erase, nor does it see the Backspace character (ASCII 8) itself.

Remember, when you type at the keyboard, the characters go to the operating system (OS). The OS normally passes those characters to your application program (e.g. to its call to **scanf()** or **cin**), but if the OS sees the Backspace character, the OS does NOT pass it on to the program. In fact, the OS doesn't pass any characters at all to the program until the user hits the Enter key.

The other mode is **raw** mode. Here the OS passes every character on to the application program. If the user hits the Backspace key, it's just treated as any other character, with no special action taken. The application program then receives an ASCII 8 character, and it's up to the programmer to decide what to do with it.

You can switch from cooked to raw mode using the **cbreak()** function, and back to cooked by calling **nocbreak()**.

Similarly, the default mode is for the OS to echo characters. If the user types the A key (and not Shift), the OS will then write 'a' to the screen.[4] There are times when we don't want echoing, such as in the examples we've seen here, and also other situations such as password entry. Well, we can turn off the echo by calling **noecho()**, and turn it back on again via **echo()**.

4 Important Debugging Notes

Don't use **printf()** or **cout** for debugging! Make sure you use debugging tool, for example GDB or better the DDD interface to GDB. If you are not using a debugging tool for your daily programming work, you are causing yourself unnecessary time and frustration. See my debugging slide show, at `http://heather.cs.ucdavis.edu/~matloff/debug.html`.

With `curses` programs, you don't really have the option of using **printf()** or **cout** for debugging anyway, as the output of those actions would become mixed with that of your application. So a debugging tool is a necessity. Here we will `curses` debugging with GDB and DDD, as they are the best known in the Unix world.[5] The problem is that you still need to do something to separate your debugging output from your `curses` application's output. Here I show how.

[4] Note, though, that even though the 'a' appears on the screen, the application programm still has not received it, which it won't do until the user hits the Enter key.

[5] This assumes prior knowledge of GDB or DDD. See the URL listed above.

4.1 GDB

Start up GDB as usual in some text window. Then choose another window in which your `curses` application will run, and determine the device name for that latter window (which we will call the "execution window"). To do this, run the Unix **tty** command in that window. Let's suppose for example that the output of the command is "/dev/pts/10". Then within GDB issue the command

```
(gdb) tty /dev/pts/10
```

We must then do one more thing before issuing the **run** command to GDB. Go to the execution window, and type

```
sleep 10000
```

Unix's **sleep** command has the shell go inactive for the given amount of time, in this example 10,000 seconds. This is needed so that any input we type in that window will be sure to go to our program, rather than to the shell.

Now go back to GDB and execute **run** as usual. Remember, whenever your program reaches points at which it will read from the keyboard, you will have to go to the execution window and type there (and you will see the output from the program there too). When you are done, type ctrl-C in the execution window, so as to kill the **sleep** command and make the shell usable again.

Note that if something goes wrong and your program finishes prematurely, that execution window may retain some of the nonstandard terminal settings, e.g. cbreak mode. To fix this, go to that window and type ctrl-j, then the word 'reset', then ctrl-j again.

4.2 DDD

Things are much simpler in DDD. Just click on View | Execution Window, and a new window will pop up to serve as the execution window.

5 Some of the Major Curses APIs, Attributes and Environment Variables

5.1 Environment

- Row and column numbers in a window start at 0, top to bottom and left to right. So, the top left-hand corner position is (0,0).
- **LINES, COLS**:

  The number of lines and columns in your window.

5.2 APIs

Here are some of the `curses` functions you can call[6] Note that the functions listed here comprise only a small subset of what is available. There are many other things you can do with `curses`, such as subwin-

[6]Many are actually macros, not functions.

dowing, forms-style input, etc.; see Section 6 for resources.

- **WINDOW *initscr()**:

  REQUIRED. Initializes the whole screen for `curses` . Returns a pointer to a data structure of type WINDOW, used for some other functions.

- **endwin()**:

  Resets the terminal, e.g. restores echo, cooked (non-cbreak) mode, etc.

- **cbreak()**:

  Sets the terminal so that it reads characters from keyboard immediately as they are typed, without waiting for carriage return. Backspace and other control characters (including the carriage return itself) lose their meaning.

- **nocbreak()**:

  Restores normal mode.

- **noecho()**:

  Turns off echoing of the input characters to the screen.

- **echo()**:

  Restores echo.

- **clear()**:

  Clears screen, and places cursor in upper-left corner.

- **move(int, int)**:

  Moves the cursor to the indicated row and column.

- **addch(char)**:

  Writes the given character at the current cursor position, overwriting what was there before, and moving the cursor to the right by one position.

- **insch(char)**:

  Same as **addch()**, but inserts instead of overwrites; all characters to the right move one space to the right.

- **mvaddstr(int, int, *char)**:

  Moves the cursor to the indicated row and column, and then writes the string to that position.

- **refresh()**:

  Update the screen to reflect all changes we have requested since the last call to this function. Whatever changes we make, e.g. by calling **addch()** above, will NOT appear on the screen until we call **refresh()**.

- **delch()**:

  Delete character at the current cursor position, causing all characters to the right moving one space to the left; cursor position does not change.

- **int getch()**:

  Reads in one character from the keyboard.

- **char inch()**:

  Returns the character currently under the cursor.

- **getyx(WINDOW *, int, int)**:

  Returns in the two **int**s the row and column numbers of the current position of the cursor for the given window.[7]

- **getmaxyx(WINDOW *, int, int)**:

  Returns in the two **int**s the number of rows and columns for the given window.

- **scanw()**, **printw()**:

  Works just like **scanf()** and **printf()**, but in a `curses` environment. Avoid use of **scanf()** and **printf()** in such an environment, which can lead to bizarre results. Note that **printw()** and **scanw()** will do repeated **addch()** calls, so they will overwrite, not insert. Note too that **scanw()** scans until it encounters a newline or Enter character, which is also true for **wgetstr()**.

- **attron(const)**, **attroff()const**:

  Turns the given attribute on or off.

5.3 Attributes

Characters can be displayed in various manners, such as normal (**A_NORMAL**) and bold (**A_BOLD**). They can be controlled via the APIs **attron()** and **attroff()**. When the former is called, all writing to the screen uses the attribute specified in the argument, until the latter is clled.

6 To Learn More

The first level of help is available in the man pages. Type

```
man curses
```

(you may have to ask for **ncurses** instead of **curses**). The individual functions have man pages too.

There are many good tutorials on the Web. Plug "curses tutorials" or "ncurses tutorials" into your favorite search engine. A tutorial you might start with is at `www.linux.com/howtos/NCURSES-Programming-HOWTO/index.shtml`.

Note, though, that some of these are tutorials for non-C usage, e.g. for Perl or Python. I don't recommend these (including my own, for Python). Not only is there different syntax but also the APIs often have some differences.

[7] Again, this is a macro, so these are **int**s rather than pointers to **int**s.