Introduction to the UNIX Curses Library

Norman Matloff
Department of Computer Science
University of California at Davis

March 16, 2003

Contents

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 **vi**; 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 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. Development of the Curses library was a major step in the evolution of UNIX software.[1]

When you log on, you have a terminal type, in the environment variable **TERM**[2]. The Curses library consists of a number of functions which your program can call. Those functions know the various cursor-movement character sequences for a large variety of terminals (this information is in the file **/etc/termcap**). The important implication of that is that your program does not have to know that information; it simply calls the Curses functions, and those functions will use your **TERM** value to check the /etc/termcap file and then send the proper cursor-movement characters.

For example, if your program wanted 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.

[1]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.

[2]You may have to set it yourself. As was mentioned before in the Syllabus and the handout on **screen**, this is done via the UNIX command (if you are using the C shell) **setenv TERM terminaltype** where **terminaltype** is vt100 for many terminals (including many terminal emulator programs running on PCs, e.g. Kermit), and is **xterm** for X Windows.

2 Some of the Major Curses APIs

Here are some of the Curses functions you can call:[3]

- **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()**:

  REQUIRED. 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 (top row is 0) and column (leftmost column is 0).

- **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.

- **refresh()**:

  Update the screen to reflect all changes we have requested since the last call to this function.

- **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.

[3]Many are actually macros, not functions.

- **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.

- **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()** (if echo is on) will do repeated **addch()** calls, so they will insert, not overwrite.

There are many other things you can do with Curses, such as subwindowing, forms-style input, etc. You can get a complete list of functions by typing

```
man curses
```

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

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
```

Below is a sample program using Curses. Its action is explained in the comments. Try running the program! You can get the source code from the raw file which produced this document, http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Curses.tex.

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

#include <curses.h>  // required

int r,c,  // current row and column
    nrows,  // number of rows in window
    ncols;  // number of columns in window

void draw(char dc)

{  move(r,c);  // move cursor to row r, column c
   delch();  insch(dc);  // replace character under cursor by dc
   refresh();  // udate 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();  // initialize window
   cbreak();  // no waiting for Enter key
   noecho();  // no echoing
   getmaxyx(wnd,nrows,ncols);  // find size of window
   clear();  // clear screen, send cursor to position (0,0)
   refresh();  // implement all changes since last refresh

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

   endwin();  // restore the original window

}

```

3 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.

Whatever approach you take to debugging (even if it is just **printf()** or **cout**), you'll need to do something to separate your debugging output from your Curses application's output. Here I show how to do that in GDB and DDD.[4]

3.1 GDB

Start up GDB as usual. Then determine the terminal number for the **execution window**, i.e. window in which you wish your Curses application to run. To do this, run the UNIX **tty** command in that window. Let's suppose for example that the output of the latter 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 **r** 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 **r** 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 them 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.

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.

3.2 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.

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