

Nurses Programming Guide

by X. Li

If you have some beginning experience in Unix programming, you may have felt the need of some text user interface operations, such as moving the cursor on the screen, editing user input, using colors, ... Such terminal IO related operations are not portable and not defined in C language. You need to either use the low-level termcap library or the curses library. Using curses/ncurses library is much easier and more portable.

Contents

Using the library.

Installations, header inclusions and link options.

Before and after using ncurses function

Thing you need to do before calling curses functions

Creating windows

Creating windows

Moving the cursor

Move the cursor around the window

Input

Read user input from keyboard

Output

Write text to the window

Using colors and other text attributes

Text colors, highlighting, blinking, ...

Using curses in Xterm

How to cope window resizing by user

An example of complete programs

A typing speed test program by me

Using ncurses library

To compile your C/C++ programs using ncurses/curses library you need to include the curses header file `<curses.h>`. For ncurses, you may include either `<curses.h>` or `<ncurses.h>`. In some systems, you must include `<ncurses.h>`.

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

To link the programs you need to use the `-lcurses` or `-lncurses` option, like

```
gcc -lncurses prog.c
```

This way the program is dynamically linked to the ncurses library. To run it in another computer, the system must have the ncurses library installed. If you want to avoid the trouble, you may have it statically linked. To do that, find the file `libncurses.a` in `/usr/lib` and do

```
gcc prog.c libncurses.a
```

Most Unix systems have curses or ncurses installed as a default option. To find out if it's installed, you can try `man ncurses` `man curses` or go to `/usr/lib` and `/usr/include` to list the files.

If you're installing Linux or FreeBSD on your own machine, be sure to install the **ncurses-development** package in order to do ncurses programming.

Initialization

The very first thing to do: Before you use any other *curses* routines, the `initscr()` routine must be called first.

```
initscr();
```

If your program is going to write to several terminals, you should call `newterm` instead, which is another story.

One-character-a-time. To disable the buffering of typed characters by the TTY driver and get a character-at-a-time input, you need to call

```
cbreak();
```

No echo. To suppress the automatic echoing of typed characters, you need to call

```
noecho();
```

Special keys. In order to capture special keystrokes like **Backspace**, **Delete** and the four arrow keys by `getch()`, you need to call

```
keypad(stdscr, TRUE);
```

Before exiting. Before the program is terminated, `endwin()` must be called to restore the terminal settings.

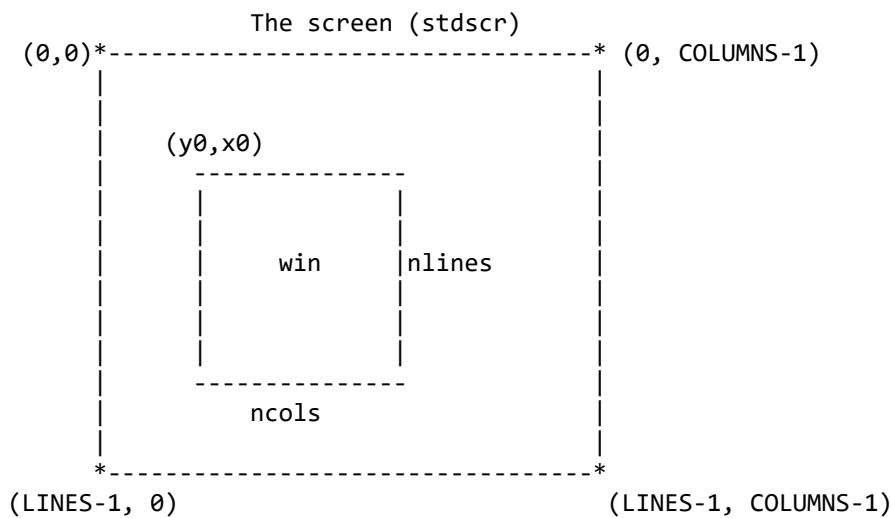
Windows

A window is a 2-dimensional array of characters representing all or part of a CRT screen. Character input and output should pertain to a specific window.

The default window. A default window called `stdscr`, which is the size of the terminal screen, is supplied. To use the `stdscr` window, you don't need to do any initializations. You can also divide the screen to several parts and create a window to represent each part.

Create a new window. The data structure of window is WINDOW, defined in ncurses.h. To declare and create a new window, do

```
WINDOW * win = newwin(nlines, ncols, y0, x0);
```



All the 4 parameters are **ints**. Here nline is the height of the window -- number of lines, ncols is the width -- number of columns of the window. y0 and x0 are the coordinates of the upper left corner of win on the screen -- line y0 and columns x0. You should make sure that the area of the new window is inside the screen.

Height and width of the window. The size of the whole screen can be determined by the two global variables COLUMNS and LINES. y0 and x0 should satisfy

```
0 <= y0 < LINES;  
0 <= x0 < COLUMNS;
```

In X window system, the actual xterm size might be changed leaving these two variables obsolete. In this case you should use the macro void getmaxyx(WINDOW *, int y, int x) to get the size of the screen.

```
int h, w;  
getmaxyx(stdscr, h, w);
```

No overlapping. Windows cannot overlap with each other. Therefore you have two options: only use stdscr and no other windows, or create several non-overlapping windows but do not use stdscr.

Refresh. If you make some change to a window, such as printing something or moving the cursor, the effect is not shown on the screen until you call the wrefresh() function

```
wrefresh(win);
```

Clear window. To erase everything written in the window win, call wrefresh(win). refresh() is equivalent to wrefresh(stdscr).

Delete window. If a window win is no longer needed, and you're going to create new windows to overlap it, you should call delwin(win) to delete the window (release the memory it is using).

Moving the cursor

The position of the cursor on the screen is important because it is default beginning place for most output functions. The cursor also shows the user where the input is expected.

To move the cursor to a new position on a window, use the function `int wmove(WINDOW *win, int y, int x)`

```
wmove(win, y, x);
```

where (x, y) are the coordinates of the new position in the window. If the window has `nlines` lines and `ncolumns` columns, then

```
0 <= y < nlines
0 <= x < ncolumns
```

Refresh. The actual cursor motion is not shown on the screen until you do a `wrefresh(win)`.

`move(y, x)` is equivalent to the `wmove(stdscr, y, x)`.

Input

To read a character from `stdscr`, use the function `int getch(void)`.

```
int ch = getch();
```

No echoing. If you have called `noecho()`, the character `ch` will not be printed on the screen, otherwise it will. Disabling automatic echoing gives you more control over the user interface.

No buffering. If you have called `cbreak(void)` each key the user hits is returned immediately by `getch()`. Otherwise the keys hit by the user are queued until a newline is read. Then calls to `getch()` take characters from the queue in FIFO manner until the queue is empty and the next whole line is read.

No delaying. Usually a call to `getch()` waits until a key is hit. If you have called `nodelay(stdscr, TRUE)`, then `getch()` will work in a non-blocking manner -- it will return `ERR` if the key input is not ready. This is usually useful for writing game-like programs, where the promptness of user response matters. For example

```
int ch;
nodelay(stdscr, TRUE);
for (;;) {
    if ((ch = getch()) == ERR) {
        /* user hasn't responded
        ...
        */
    }
    else {
        /* user has pressed a key ch
        ...
        */
    }
}
```

Special keys. If you have called `keypad(stdscr, TRUE)`, then if the user hits a special key such as the **Delete** key, the arrow keys, **Ctrl** combined keys and function keys, a single int value will be returned. Here is the definition of several special keys

key code	description
KEY_DOWN	The four arrow keys ...

KEY_UP	
KEY_LEFT	
KEY_RIGHT	
KEY_HOME	Home key
KEY_BACKSPACE	Backspace
KEY_F(n)	Function keys, for $0 \leq n \leq 63$
KEY_DC	Delete character
KEY_IC	Insert char or enter insert mode
KEY_ENTER	Enter or send

For a complete list read the man page of `getch()`.

Catch special keys. To use these keys, you need to check the return value of `getch()`. For example

```
int ch = getch();
switch (ch) {
    case KEY_BACKSPACE: /* user pressed backspace */
        ...
    case KEY_UP: /* user pressed up arrow key */
        ...
    case KEY_DOWN: /* user pressed up arrow key */
        ...
    case 'A' .... /* user pressed key 'A' */
        ...
}
```

Read character from a window. The function `int wgetch(WINDOW *win)` reads a key from a window. The user input of course comes from the keyboard and not the screen window. But the different windows on the screen might have different delay modes and other properties, therefore affect the behavior of `wgetch()`.

Moving the cursor and read a character. There are also functions which combine cursor moving and character reading together

```
int mvwgetch(int y, int x);
int mvwgetch(WINDOW *win, int y, int x);
```

Output

The function `int waddch(WINDOW *win, chtype ch)` adds a character on the window at the current cursor position, and the cursor position is advanced then.

Wrap. If the new position of the cursor is out of the window, it wraps to the beginning of the next line.

Scroll. If the next line is out of the window, and you have called `scrollok(win, TRUE)` when the window was created, the stuff in the window is scrolled up one line.

Character attribute. The parameter `ch` is of type `chtype()`, which is the ASCII value of the character combined with some video attributes such as colors. The combination is through the logical **OR** of the character value and the attribute, which I will talk about in the section.

Refresh. After a call to `waddch`, the screen is not updated until you call `wrefresh(win)`.

Other output functions

`mvwaddch(win, y, x, ch);` is equivalent to `wmove(win, y, x); waddch(win, ch);`

`addch(ch);` is equivalent to `waddch(stdscr, ch);`.

`wechochar(win, ch);` function and `echochar(ch)` are equivalent to `waddch(win, ch); wrefresh(win);` and `addch(ch); refresh();` respectively. But `echochar` and `wechochar` may be more efficient.

`int waddstr(WINDOW *win, const char *str)` and `int addstr(const char *str)` prints a null-terminated string at the cursor position of the window, and advance the cursor position accordingly.

The functions `int wprintw(WINDOW *win, char *fmt ...)` and `int printw(char *fmt ...)` do formatted output in the same fashion as the analogous standard library function `printf`.

Attribute

When characters are drawn on the screen some special video effects, like *foreground and background color, highlight, underline, blinking*, ..., can be shown. Such video effects are represented by integers called text attributes. Each significant bit of the attribute corresponds to one video effect.

Using attribute

There are two ways to use attribute. One is by passing `waddch(win, ch)` a character value combined with attribute. The other is setting the global window attribute.

Character type. When calling `waddch(win, ch)` or `addch(ch)`, logical OR the character value with the attribute. For example, `A_UNDERLINE` is the predefined attribute for underlining. To print the character 'X' with underlining, do

```
waddch(win, 'X' | A_UNDERLINE);
```

Using several attributes is of course possible. For example, to print the character 'X' with highlight in color pair 3

```
waddch(win, 'X' | A_UNDERLINE | COLOR_PAIR(3));
```

Setting window attribute. `int wattron(WINDOW *win, int attr)` function to turn on an attribute `attr`. Then anything printed by subsequent calls to `waddch`, `addstr` and `waddstr` will have the attribute `attr`, For example, to print a highlighted message on the screen

```
attron(A_STANDOUT);  
addstr("I am highlighted!\n");
```

Predefined attributes. Here is some attributes defined in `ncurses.h`

<code>A_NORMAL</code>	Normal display (no highlight)
<code>A_STANDOUT</code>	Best highlighting mode of the terminal.
<code>A_UNDERLINE</code>	Underlining
<code>A_REVERSE</code>	Reverse video
<code>A_BLINK</code>	Blinking
<code>A_DIM</code>	Half bright
<code>A_BOLD</code>	Extra bright or bold
<code>A_PROTECT</code>	Protected mode
<code>A_INVIS</code>	Invisible or blank mode
<code>A_ALTCHARSET</code>	Alternate character set
<code>A_CHARTEXT</code>	Bit-mask to extract a character
<code>COLOR_PAIR(n)</code>	Color-pair number n

Using colors

The combination of foreground and background color is an attribute. Unlike other attributes, before using colors, you must call `start_color()`.

When `start_color()` is called, a set of *colors* and *color pairs* are created which you can use. The number of available colors and the number of the color pairs are stored in two global variables `COLORS` and `COLOR_PAIRS`. To use an predefined color pair as an attribute, you need to call the macro `COLOR_PAIR(n)`, where `n` must satisfy

$$0 \leq n < \text{COLORS}$$

Example. To give a window the color attribute defined by color pair #2, so that each subsequent character printed in this window has the foreground and background color defined by color pair #2

```
wattron(win, COLOR_PAIR(2));
```

The meaning of a color pair can be redefined. For example

```
init_pair(1,2,0);
```

redefine the color pair #1 with foreground color #2 and background color #0. In the function `int init_pair(short n, short f, short b)` the parameters must satisfy

$$\begin{aligned} 0 &\leq n < \text{COLORS} \\ 0 &\leq f < \text{COLOR_PAIRS} \\ 0 &\leq b < \text{COLOR_PAIRS} \end{aligned}$$

When `start_color()` is called, 8 basic colors are initialized

```
COLOR_BLACK  
COLOR_RED  
COLOR_GREEN  
COLOR_YELLOW  
COLOR_BLUE  
COLOR_MAGENTA  
COLOR_CYAN  
COLOR_WHITE
```

You can use these names in `init_pair()` for specifying foreground and background color.

To find out what foreground color and background color is used by a color pair, use the function `int pair_content(short pair, short *f, short *b)`. To find out the definition of a color use the function `int color_content(short color, short *r, short *g, short *b)`

Color can also be redefined by `int init_color(short n, short r, short g, short b)`, where `n` is the index of color, must be less than `COLORS`. `r`, `g`, and `b` represent the intensity of red, green and blue. Each value of `r`, `g` and `b` must be less than 1000.

Line graphics

Line graphics. Here are some special characters which can be used in `addch` and `addstr` routines as the `chtype`.

<code>ACS_BLOCK</code>	solid square block
<code>ACS_BOARD</code>	board of squares
<code>ACS_BTEE</code>	bottom tee
<code>ACS_BULLET</code>	bullet
<code>ACS_CKBOARD</code>	checker board (stipple)
<code>ACS_DARROW</code>	arrow pointing down
<code>ACS_DEGREE</code>	degree symbol
<code>ACS_DIAMOND</code>	diamond

ACS_GEQUAL	greater-than-or-equal-to
ACS_HLINE	horizontal line
ACS_LANTERN	lantern symbol
ACS_LARROW	arrow pointing left
ACS_LEQUAL	less-than-or-equal-to
ACS_LLCORNER	lower left-hand corner
ACS_LRCORNER	lower right-hand corner
ACS_LTEE	left tee
ACS_NEQUAL	not-equal
ACS_PI	greek pi
ACS_PLMINUS	plus/minus
ACS_PLUS	plus
ACS_RARROW	arrow pointing right
ACS_RTEE	right tee
ACS_S1	scan line 1
ACS_S3	scan line 3
ACS_S7	scan line 7
ACS_S9	scan line 9
ACS_STERLING	pound-sterling symbol
ACS_TTEE	top tee
ACS_UARROW	arrow pointing up
ACS_ULCORNER	upper left-hand corner
ACS_URCORNER	upper right-hand corner
ACS_VLINE	vertical line

Usually on terminals using these symbols can draw pretty windows and shapes. One place to use this is the `wborder` function, which draws borders for a window. See the man page for details about the parameters, but usually do it the following way

```
wborder(win, 0, 0, 0, 0, 0, 0, 0, 0);
```

will make a good looking window.

To draw a horizontal line across the window, do

```
whline(win, ACS_HLINE, ncolumns);
```

X window

If an xterm is resized the contents on your text windows might be messed up. To handle this gracefully you should redraw all the stuff based on the new height and width of the screen. When resizing happens, your program is sent a **SIGWINCH** signal. You should catch this signal and do redrawing accordingly. Here is some hint.

```
#include <signal.h>
void* resizeHandler(int);

int main(void) {
    ...
    signal(SIGWINCH, resizeHandler);
    ...
}

void* resizeHandler(int sig)
{
    int nh, nw;
    getmaxyx(stdscr, nh, nw); /* get the new screen size */
    ...
}
```