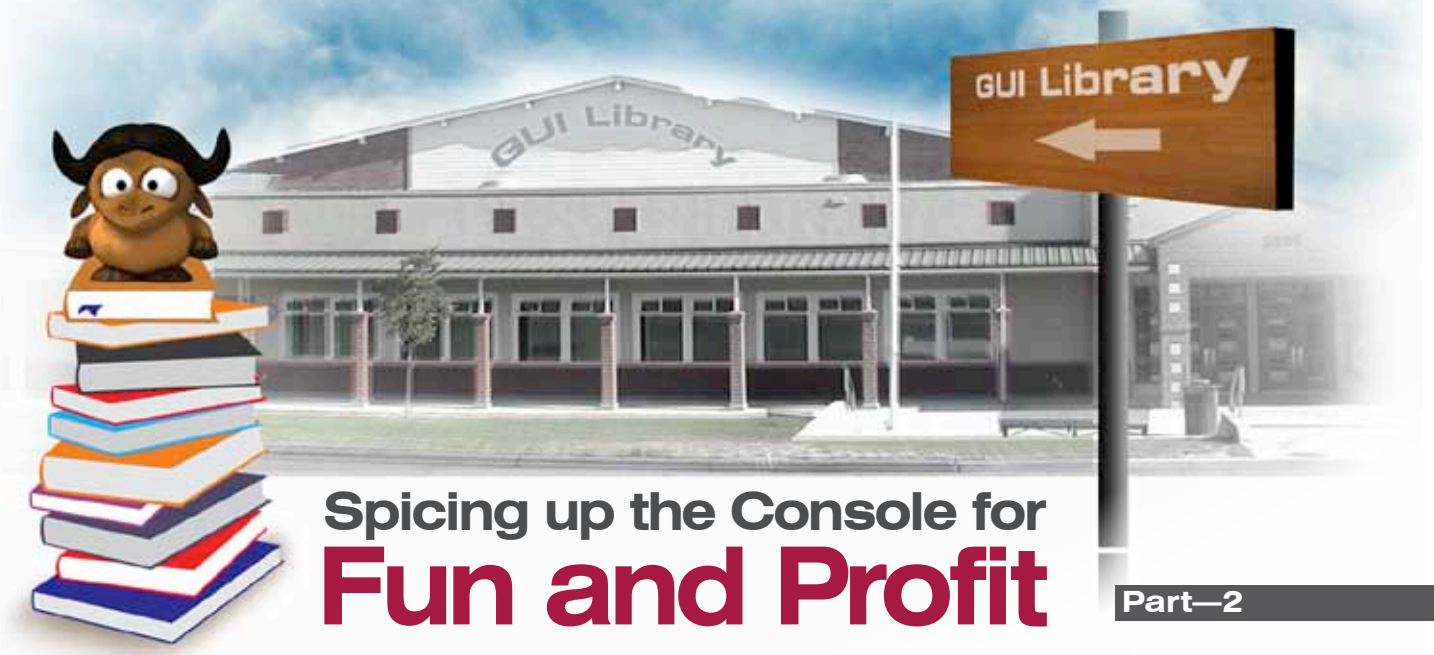

Spicing up the Console for Fun and Profit

Part—2

In the previous part of this article, we manipulated various aspects of the shell and terminals, like prompts, colours, cursor movements, etc, using escape sequences and utilities like tput. Now let's move to the programming libraries used to create sophisticated and portable console GUIs. First, let us explore some of the underlying concepts of these tool-kits, and then move to hands-on sections on the newt and ncurses terminal GUI libraries.

Terminals and their databases

In the first part of this article, we used mostly raw escape sequences to manipulate text terminals, from setting colours to moving the cursor around. Now, let us go deeper into the very close relationship between console terminals and the escape/control sequences. In the old days, text terminals were Teletype hardware devices, each with its own command sequences to perform various operations. Due to this difference, not all command sequences are guaranteed to work with all terminals. This is equally applicable to emulated terminals or text consoles that we have in the modern GNU/Linux world.

When this is the case, how do we write console GUI applications that behave uniformly, irrespective of terminal differences? This need was felt long ago, and the result was the terminal database. A terminal database stores information about the capabilities of terminals, and the related command sequences in order to manipulate them in a uniform way. Thus, you can write console GUI applications that behave the same way whether you run them in Xterm or a GNU/Linux console. The *tput* utility we used in the first part of the article is an example that uses the terminal database internally. In fact, all terminal GUI libraries like *newt*, *curses*, *ncurses*, etc, manipulate terminal databases internally, to provide an API to create portable terminal applications.

The first terminal database library was *termcap*, and later on it was enhanced in the form of *terminfo*, written in the 80s for pcurses. Currently, *terminfo* is the most preferred terminal database and library used for text terminals, and it also provides an emulated *terminfo* for the older console applications. If you install *ncurses*, as we shall do for the *ncurses* section, then that comes with *terminfo* and its terminal database compiler, known as *tic*. The *terminfo* database is stored on disk in a compiled form. The capabilities and control sequences for various terminals are arranged as tables corresponding to Boolean, string and numeric capabilities. The various capabilities information (like screen operations, scrolling padding requirements, terminal initialisation sequences, etc) is accessed through the *terminfo* library functions. This is a vast topic in itself, and you could refer to the *ncurses* documentation and *terminfo* man page to explore it further.

Easy console GUIs with *newt*

Now it's time to get our hands dirty with *newt*, a terminal GUI tool-kit. *Newt* started its life at Red Hat when there was a need for a light-weight, no-fuss and stable terminal tool-kit for the installation process. We might all have interacted with *newt* knowingly or unknowingly—it is the tool-kit used by the *Anaconda* installer used in RHEL, Fedora and CentOS distributions. The name stands for *Not Erik's Windowing Toolkit*, and it is based upon the *slang* library. As the installer code runs in a restricted environment, it was developed using the C programming language.

Newt is different from many event-based GUI tool-kits, as

its original goals were different from modern tool-kits where there is an event loop waiting to invoke callbacks registered for the various events. It is a *serialised* model-based tool-kit, where various windows are constructed in a stack fashion, and the latest one is always the active one. You can understand the behaviour of *newt* windows as model windows, where only the topmost (the latest created) window is active, until you pop it to destroy it. So you can use *newt* in applications where the flow of execution is strictly serial, like most of the candidates for scripting. *Newt* is very easy to program (one of its original goals), and you'll be productive right away, after playing with the example code presented in this section.

You need to have the *newt* library and development headers installed on your system, along with build tools like GCC, to follow the examples in this section. The good news is that *newt* comes pre-installed in most cases (Ubuntu, Fedora and CentOS). Also, there is a utility known as *whiptail*, based on *newt*, which is pre-installed on these distros. This utility adds some basic dialogue boxes to the shell, and other scripting languages where you can call external commands. I encourage you to explore its man page.

If you see usage information on running *whiptail* in a console, then you only have to install *newt* development headers (as the super-user, and run *apt-get install libnewt-dev*). If you don't have a development environment set up on your machine, do that with *apt-get install build-essential* (as the super-user). There is also a Python module for *newt*, known as *snack*; you can install it with *apt-get install python-newt* to run the Python examples presented in this section. In fact, you can also program *newt* with Perl, PHP and Tcl. You can explore those on your own, through the links provided in the *newt* Wikipedia page, if interested.

I'll explain *newt*' s basic programming model, and then we'll go on to programming examples, with specific notes where applicable. *Newt* provides functionality to create various forms or windows, widgets or controls. All *newt* windows are drawn on a special background window, known as the root window. You can put various controls on the root window itself, but in most cases, these are placed on various forms to provide the desired functionality. You can't randomly control which window is to be shown; only the latest-created window is active. You have to 'pop' windows in the reverse order of creation, to destroy them and return to the previous window.

Now compile *testnewt.c* as shown below, with *gcc -o testnewt testnewt.c -lnewt* (after changing the working directory to where *testnewt.c* is saved). Run it with *./testnewt* and terminate it by pressing any key. (Note that source code files can be downloaded from *www.linuxforu.com/articles_source_code/nov11/spice_codes.zip*)

```
#include <newt.h>
#include <stdlib.h>
#include <string.h>
```

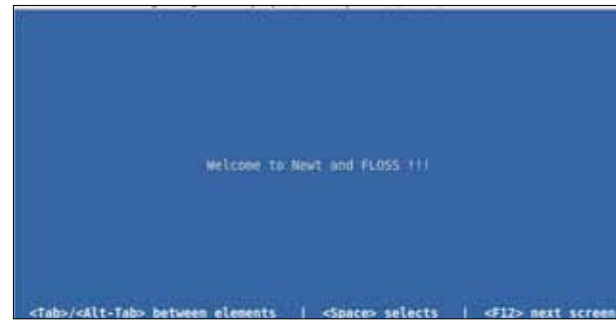

Figure 1: Newt standard screen with message

```
int main(int argc, char* argv[]) {
    /* required variables and string */
    unsigned int uiRows, uiCols;
    const char pcText[] = "Welcome to Newt and FLOSS !!!";

    /* initialisation stuff */
    newtInit();
    newtCls();

    /* determine current terminal window size */
    uiRows = uiCols = 0;
    newtGetScreenSize(&uiCols, &uiRows);

    /* draw standard help and string on root window */
    newtPushHelpLine(NULL);
    newtDrawRootText((uiCols-strlen(pcText))/2, uiRows/2, pcText);

    /* cleanup after getting a keystroke */
    newtWaitForKey();
    newtFinished();
    return 0;
}
```

As is evident in the source, *newt* allocates and initialises internal data structures, and sets the terminal to the proper state by *newtInit*. The *newtGetScreenSize* function calculates the columns and lines of the terminal screen. A text string is put on the background by providing desired column, line and content to *newtDrawRootText*. Also, a *newt* standard help string is put at the bottom of the root window by calling *newtPushHelpLine* and passing a NULL parameter to it. If you pass a string to this function, then that string is displayed as *help* content. The *newtWaitForKey* function blocks the program till any key is pressed—and finally, *newtFinished* clears *newt* internal data structures, and resets the terminal to its original state. If you don't call *newtFinished* at the end of your program, then the user will have to restore the original settings via the *reset* command. The output of the program on Ubuntu is shown in Figure 1.

The next example creates a console-mode rectangle animation through window creation and destruction. If you don't pop a created window, then the next one is overlapped on the top of the older one. The *animaterect.c* code takes the

Figure 2: Windows overlapping and clean-up in *newt*

following parameters: the first, the number of animation steps; the second, a non-zero value if you want to erase the previous windows during the animation. Build and run it; and you can see a rectangular window moving diagonally, on pressing any key multiple times. The output of the program, in both clean-up and overlapping modes, is shown in Figure 2. Remember, you can create various help content by adding different help lines. The help-line stack is different from the windows stack, so you can control them independent of each other.

```
#include <newt.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void cleanupNewt() {
    newtFinished();
}

void initNewt() {
    newtInit();
    newtCls();
    newtDrawRootText(1, 1, " Welcome to Animation with Newt
!!!");
    newtPushHelpLine(" < Press any key to animate the rectangle
> ");
}

/* draw rectangle with specified title, width & height */
void drawRect(unsigned int uiX, unsigned int uiY, unsigned int
uiW, \
     unsigned int uiH, const char* pTitle, unsigned int bErase) {
    newtOpenWindow(uiX, uiY, uiW, uiH, pTitle);
    newtWaitForKey();
    if(bErase) {
        newtPopWindow();
    }
}

/* Main routine for the rectangle animation */
int main(int argc, char* argv[])
{
    /* constant and variable data required */
    char pRectTitle[20];
    unsigned int uiWidth = 0, uiHeight = 0, uiMaxSteps = 0;
    unsigned int uiRectWidth = 0, uiRectHeight = 0, uiEraseFlag
= 0;

    /* check for command line parameters */
    if(3 != argc) {
        printf(" Usage: animaterect maxsteps eraseflag\n");
        return -1;
    }
    else {
        uiMaxSteps = atoi(argv[1]);
        uiEraseFlag = atoi(argv[2]);
        if(2 > uiMaxSteps) {
            printf(" Error: minimum value of maxsteps should be
2.\n");
            return -1;
        }
    }
    initNewt();

    /* calculate dimensions of rectangles to accomodate in the
terminal area */
    newtGetScreenSize(&uiWidth, &uiHeight);
    uiRectHeight = (uiHeight - 6)/uiMaxSteps;
    uiRectWidth  = (uiWidth - 6)/uiMaxSteps;

    int i;
    for(i=0; i < uiMaxSteps; ++i) {
        snprintf(pRectTitle, 19, "Rectangle %d", i+1);
        drawRect(3+i*uiRectWidth, 3+i*uiRectHeight, \
      uiRectWidth, uiRectHeight, pRectTitle, uiEraseFlag);
    }
    cleanupNewt();
```

Figure 3: *Newt* form with widgets

```
    return 0;
}
```

Now, it's time to use *newt* widgets and forms to create some terminal GUI examples. Compile and run *greeter.c*, which uses the *Label*, *Entry and Button* components. Figure 3 shows the resultant form.

```
#include <newt.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void cleanupNewt() {
    newtFinished();
}

void initNewt(unsigned int y, const char* pTitle) {
    unsigned int uiWidth = 0, uiHeight = 0;
    newtGetScreenSize(&uiWidth, &uiHeight);
    newtInit();
    newtCls();
    newtDrawRootText((uiWidth-strlen(pTitle))/2, y, pTitle);
    newtPushHelpLine(" < Press ok button to see your greetings >
");
}

/* draw centered window with components and specified title */
void drawWindow(unsigned int uiW, unsigned int uiH, char* pTitle,
char* pEntryText) {
    newtComponent form, label, entry, button;
    char* pEntry;
    newtCenteredWindow(uiW, uiH, pTitle);
    label = newtLabel(2, uiH/4, "Geek's Name");
    entry = newtEntry(uiW/2, uiH/4, "RichNusGeeks", 12, \
                      (const char**) &pEntry, 0);
    button = newtButton((uiW-6)/2, 2*uiH/3, "Ok");
    form = newtForm(NULL, NULL, 0);
    newtFormAddComponents(form, label, entry, button, NULL);
    newtRunForm(form);
    strncpy(pEntryText, pEntry, 12);
    newtFormDestroy(form);
}
```

```
/* draw a centered message box */
void messageBox(unsigned int uiW, unsigned int uiH, const char*
pMessage) {
    newtComponent form, label, button;
    newtCenteredWindow(uiW, uiH, "Message Box");
    newtPopHelpLine();
    newtPushHelpLine(" < Press ok button to return > ");
    label = newtLabel((uiW-strlen(pMessage))/2, uiH/4, pMessage);
    button = newtButton((uiW-6)/2, 2*uiH/3, "Ok");
    form = newtForm(NULL, NULL, 0);
    newtFormAddComponents(form, label, button, NULL);
    newtRunForm(form);
    newtFormDestroy(form);
}

int main(int argc, char* argv[]) {
    /* constant and variable data required */
    const char pBackTitle[] = "-----Newt Greeter for FOSS
Hackers-----";
    char pName[20], pMessage[40];

    initNewt(1, pBackTitle);
    drawWindow(30, 10, "Greeter !!!", pName);
    snprintf(pMessage, 39, "FOSS loves geeks like %s \\m/",
pName);
    messageBox(40, 10, pMessage);
    cleanupNewt();
    return 0;
}
```

The greeter program presents some new functions—let me explain. A *newt* form is meant to group components logically. All *newt* components (widgets), including forms, are represented by a data structure *newtComponent*. The basic flow of actions when creating components is as follows:

1. Create a container window with *newtOpenWindow* or *newtCenteredWindow*. If you skip this, then your widgets are drawn on the last window left, which is confusing and visually distracting in most cases.
2. Create a new form (*newtForm*) for every logical component grouping.
3. Create the components to put in the form, and add them with *newtFormAddComponents*.
4. Run the form (*newtRunForm*) to display it on-screen.
5. Store all return values from the components for further handling.
6. Finally, free the resources allocated for the form (and sub-forms) and components by calling *newtFormDestroy*.

You can thus group forms and sub-forms to create sophisticated and feature-rich terminal GUIs. You can learn about the arguments to various *newt* functions via the tutorial link in *References*.

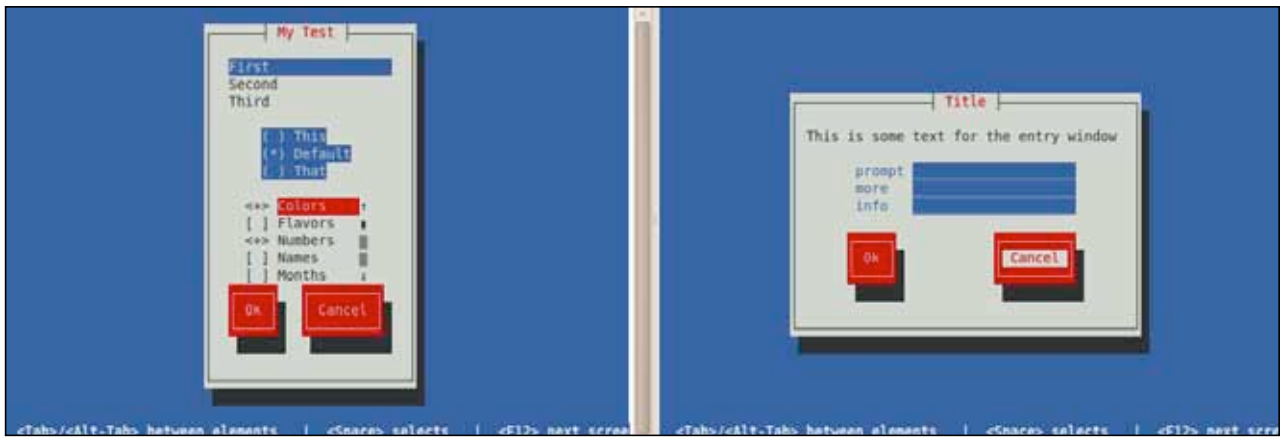

Figure 4: *Snack* example applications

It's easier to program *newt* with the Python module, *snack*. In fact, the Red Hat installer, *Anaconda*, actually uses *newt* through *snack*, and is the best real-world example of this. In Ubuntu, you'll find two sample programs, *peanuts.py* and *popcorn.py* in */usr/share/doc/python-newt/examples/*, which quickly demos *snack* in action, as seen in Figure 4. It should be easy to explore *snack* with your knowledge of *newt*, and the tutorial linked in References. *Newt* programming is a cakewalk with *snack*, so I conclude this section leaving a few trivial newt concepts for you to explore yourself.

More sophisticated console GUIs with *ncurses*

Now I come to the grand-daddy of all terminal GUI tool-kits—*ncurses*, which is the most sophisticated, powerful and seasoned terminal tool-kit, and is still very much in active use in the *NIX world. Look it up at Wikipedia and you will be amazed with its abilities and modern-day uses. From the text-mode configuration utility for the Linux kernel, to GNU Screen, to the Aptitude front-end for Debian package management, ncurses is everywhere. The very popular Dialog program (covered in detail in the April 2010 issue of LFY) to create terminal GUIs is also based on *ncurses*. This section is a fast-track hands-on session, as detailed *ncurses* programming is a book's worth of material. *Ncurses* has bindings for more than a dozen programming languages, and also some frameworks like CDK and NDK++, to make programming it as easy as possible. As *curses* is a standard module available with Python, I'll use mainly Python for example code, with some C for basic concepts.

Ncurses is pre-installed on most distributions; you have to install headers only for C/C++ development (run *apt-get install ncurses-dev* in Ubuntu, as the super-user). You can also build it from the sources (requires a C and C++ development set-up) following the official documentation. Build *testncurses.c* with *gcc -o testncurses testncurses.c -lncurses* and run it to quickly see *ncurses* in action.

```
#include <ncurses.h>
#include <string.h>

void init() {
    initscr();
    raw();
    keypad(stdscr, TRUE);
    noecho();
}

void cleanup() {
    getch();
    endwin();
}

/* messages printing */
void prompt(unsigned int uiY, unsigned int uiX, const char* pStr)
{
    mvprintw(uiY, uiX, pStr);
}

int main(int argc, char* argv[]) {
    /* required data */
    unsigned int uiH = 0, uiW = 0;
    const char pPrompt[] = "Enter the Geeky password : ";
    const char pHelp1[]  = "< Type correct password to see your
messages >";
    const char pHelp2[]  = "< Press any key to leave
>                    ";
    const char pHelp3[]  = "< Incorrect password, press any key
to leave >";
    const char pMsg1[]   = "When men were men and wrote their
device drivers";
    const char pMsg2[]   = "and said yes only to Free Open Source
Software.";
    char pPasswd[20];
    init();

    /* show the initial messages */
    getmaxyx(stdscr, uiH, uiW);
    prompt((uiH-2), 2, pHelp1);
    prompt(uiH/2, (uiW-strlen(pPrompt))/2, pPrompt);
```

```
    getstr(pPasswd);

    /* valid/invalid password handling */
    if(!strcmp(pPasswd, "lfyrockz")) {
        prompt((uiH-2), 2, pHelp2);
        prompt((uiH/2)-1, (uiW-strlen(pMsg1))/2, pMsg1);
        prompt(uiH/2, (uiW-strlen(pMsg2))/2, pMsg2);
    }
    else {
        prompt((uiH-2), 2, pHelp3);
    }
    refresh();
    cleanup();
    return 0;
}
```

Now, I will explain the real intent behind the various functions used in *testncurses.c*, and a few other basic *ncurses* blocks. The *initscr* function is to initialise the terminal for *curses* mode, and allocates the resources for various data structures. *Ncurses* programs always draw on screen areas known as *windows*. These windows are an abstract concept, and useful to group various terminal screen operations related to one another. The various windows/screens combine to make a complete interactive console application. The *stdscr* is the default window provided by *ncurses*—the first screen you encounter after running terminal programs. All operations to the windows are performed in memory, and then these are reflected on the terminal using the refresh function. This way, *ncurses* minimises internal operations, and only refreshes the terminal screen when windows change. You get the current terminal dimensions using the *getmaxyx* macro. The raw disables line buffering, so that each character is available to your program as entered (otherwise everything is buffered till a new line (Enter) is pressed). The *mvprintw* is used to print formatted strings to the specified screen location. Remember, all *ncurses* functions take the y coordinate before *x* when you specify locations. GUI applications use function keys like F1, F2, right/left arrows, etc, and ncurses provides this functionality through the *keypad* function. You can turn off echoing of typed characters with *noecho*, and can turn it on with *echo*. Finally, reset your terminal to the original state using *endwin*. This function also cleans up the various resources grabbed by the library. If you forget this step, the terminal goes into a weird state, and you have to reset it. If this entire sequence seems complicated, don't worry; learn it once, and use it, with a few tweaks in every *ncurses* program.

Now we move to some more appealing programs—but this time, I'll use the Python standard library module *curses*. Run *colorful.py* with *python colorful.py*. The output of the program is shown in Figure 5.

Figure 5: Colours and windows in *ncurses*

```
#! /usr/bin/env python

import curses as cur

colors = {'Black'  : cur.COLOR_BLACK,
          'Blue'   : cur.COLOR_BLUE,
          'Cyan'   : cur.COLOR_CYAN,
          'Green'  : cur.COLOR_GREEN,
          'Magenta': cur.COLOR_MAGENTA,
          'Red'    : cur.COLOR_RED,
          'White'  : cur.COLOR_WHITE,
          'Yellow' : cur.COLOR_YELLOW }

attrs = {'Bold'      : cur.A_BOLD,
         'Normal'    : cur.A_NORMAL,
         'Reverse'   : cur.A_REVERSE,
         'Underline' : cur.A_UNDERLINE, }

menu = ('1. ls -lhrt', '', '2. df -h', '',
        '3. uname -a', '', '4. hostname')

# setup the color pairs.
def initColors():
    cur.init_pair(1, colors['Red'], colors['Blue'])
    cur.init_pair(2, colors['Green'], colors['Black'])
    cur.init_pair(3, colors['Cyan'], colors['Black'])
    cur.init_pair(4, colors['Magenta'], colors['Green'])
    cur.init_pair(5, colors['Red'], colors['White'])
    cur.init_pair(6, colors['Cyan'], colors['Black'])

# stdscr window drawing routine.
def drawMainWnd(width, height, stdscr):
    stitle = 'Action Menu with Python curses module'
    shelp  = '< Use up/down arrow keys to navigate >'
    stdscr.bkgdset(' ', cur.color_pair(5))
    stdscr.border('*', '*', '*', '*', '*', '*', '*', '*');
    stdscr.addstr(2, (width-len(stitle))/2, stitle,
                  cur.color_pair(1) | attrs['Bold'] |
attrs['Underline'])
    stdscr.addstr(height-3, 2, shelp,
                  cur.color_pair(2) | attrs['Bold'])
```

```
# auxiliary window drawing routine.
def drawAuxWnd(strtx, strty, width, height, border = '#'):
    rwnd = cur.newwin(height, width, strty, strtx)
    rwnd.bkgdset(' ', cur.color_pair(6))
    rwnd.border(border, border, border, border,
                border, border, border, border)
    return rwnd

# menu in an auxiliary window.
def menuAuxWnd(rauxwnd):
    (h, w) = rauxwnd.getmaxyx()
    y = (h-len(menu))/2
    for i in menu:
        rauxwnd.addstr(y, 2, i, cur.color_pair(3) |
attrs['Bold'])
        y += 1

# output in an auxiliary window.
def outAuxWnd(rauxwnd):
    sout = "Just a Demo output!"
    (h, w) = rauxwnd.getmaxyx()
    y = h/2-1
    x = (w-len(sout))/2
    rauxwnd.addstr(y, x, sout, cur.color_pair(5))

# the main routine.
def draw(stdscr):
    cur.curs_set(0)
    initColors()
    (h, w) = stdscr.getmaxyx()
    drawMainWnd(w, h, stdscr)

    rmenu = drawAuxWnd(4, 4, (w-16)/2, h-8)
    menuAuxWnd(rmenu)
    rout  = drawAuxWnd((w+8)/2, 4, (w-16)/2, h-8, '$')
    outAuxWnd(rout)
    stdscr.refresh()
    rmenu.refresh()
    rout.refresh()
    cur.napms(5000)
    cur.flash()
    stdscr.getch()

# auto initialisation and cleanup.
cur.wrapper(draw)
```

If you are surprised by not encountering the *ncurses* initialisation and clean-up sequences seen earlier, thank Python—the Python *ncurses* module includes initialisation and clean-up functions in a class known as *wrapper*, which takes care of proper initialisation, along with *ncurses* colour support, on terminals that support colours. Also, it properly cleans up in case of any exceptions, so your terminal is left in a sane state. The wrapper class takes a function object that is responsible for your curses logic, and you could also pass it a parameter tuple if you want to supply parameters to the function. The above example creates various combinations of foreground and background colours. Please note that you cannot initialise the 0 colour pair, as that is reserved by *ncurses* for white on black background.

In Python, the string output function *addstr* combines the functionality of otherwise multiple *ncurses* functions for the purpose. You can create various windows through *newwin* and set their background styles using *bkgdset*. Also, there is a *border* function to customise the rectangular boundaries surrounding the windows. I also used *napms* to introduce a few milliseconds' delay in the program, and *flash* to blink the terminal screen for notification. You can see that *stdscr* is also a kind of window, and how easily you can control various parts of the GUI screen by using separate windows.

This section should load you with the required *ncurses* knowledge to explore more complicated stuff like panels, menus and forms. The *ncurses* sources come with a whole bunch of great examples and documentation. Browse through those to learn more and progress towards sophisticated but lightweight and very stable terminal applications.

Console terminal-based GUI tool-kits have been going strong for decades, and there is a plethora of modern console applications that use them. Being lightweight and highly stable were some of the original goals of tool-kits like newt and ncurses. The *NIX ecosystem, especially servers, embedded systems, legacy machines, etc, are still very dependent on less resource-hungry and powerful terminal-based configuration utilities for day-to-day operations. You can create very light, stable, portable and innovative terminal GUIs using these tool-kits, with minimal effort. END

References

- Newt library tutorial: *http://gnewt.sourceforge.net/tutorial.html*
- Quick guide to Python snack module: *http://www.wanware.com/tsgdocs/snack.html*
- Ncurses home page: *http://www.gnu.org/software/ncurses/ncurses.html*
- Writing programs with ncurses: *http://invisible-island.net/ncurses/ncurses-intro.html*
- Python standard library documentation for curses: *http://docs.python.org/library/curses.html*

By: Ankur Kumar Sharma

The author is a software developer and researcher, who feels he is highly indebted to FOSS, rock music, Zen, Ancient Sciences and all other mysterious things that keep him on his toes. You could find his other FOSS stuff on www.richnsgeeks.com and he looks forward to your valuable feedback.