CHAPTER 1

The Setup

This chapter covers a basic setup and organization for you to get started with NCurses programming. Here you'll find:

- An introduction to the terminal window in UNIX
- A smattering of basic shell commands
- Creating a special `curses` directory for this document's programs
- A review of available text editors
- The creation of a basic NCurses program
- A review of the gcc compiler and linking commands
- Re-editing source code and debugging exercises

The idea here is to show you how everything works and to get you comfortable programming with NCurses, even if you've never written a UNIX program before.

NCurses Is a UNIX Thing

You must have a UNIX-like operating system to work the samples and examples in this book.

Beyond this, note that you must also have the programming libraries installed for your operating system. Without those libraries, programming in NCurses just isn't gonna happen. Refer to your operating system's installation or setup program, such as `/stand/sysinstall` in FreeBSD, to install the C programming libraries for your operating system. If special extensions are required to get the NCurses library installed, use them!

NOTE **It's possible to program NCurses in Windows when using the Cygwin environment. I've not toyed with Cygwin, so I'm unable to comment on it here. For more information, refer to** `www.cygwin.com`**.**

Run (Don't Walk) to a Terminal Screen Near You

NCurses is about programming the terminal screen, so you'll need access to a terminal screen or window to run the programs.

You can either use one of the virtual terminals (which you can access on most PCs by pressing Alt+F1, Alt+F2, Alt+F3, and so on) or open a terminal window in the X Window System environment or in Mac OS X using the Terminal program. (See Figure 1-1.)

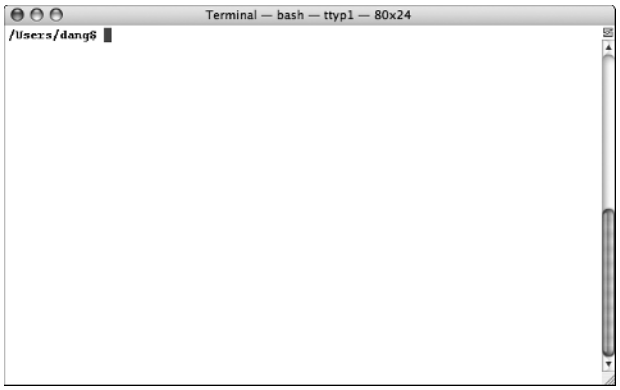

Figure 1-1: A terminal window for Mac OS X

Note that the terminal you choose can affect what NCurses does. Not all terminal types can, for example, do color or draw lines on the screen.

Know Something About the Shell

The program you use in the terminal screen is a *shell*. It displays a shell prompt and lets you type one of the gazillions of UNIX commands and what not — which is all basic UNIX stuff.

The following sections review basic shell operations and a smattering of commands. If you feel you already know this, skim up to the section titled "Make a Place for Your Stuff."

Some Shelly Stuff

For example, the standard Bourne shell may look like this:

```
$
```

The dollar sign is the prompt, and you type your commands after the prompt.

The Bash shell, popular with Linux, may look like this:

```
Bash-2.05a$
```

Or the shell may be customized to display your login name:

```
dang$
```

Or even the working directory:

```
/home/dang/$
```

Whatever!

No one really cares about which shell you use, but you should know enough shell commands to be able to do these things:

- Make directories
- Display a file's contents
- Copy files
- Rename files
- Remove files

It's beyond the scope of this book to teach you such stuff, though a handy list of popular shell commands is provided at the end of this chapter.

Note that this book does not display the shell prompt when you're directed to enter a command. Simply type the command; then press Enter to send the command to the shell program for processing.

It is always assumed that you press the Enter key to input the command.

NOTE Please do check your typing! The shell is very fussy about getting things correct. In the Bash shell, you'll see a `command not found` error when you mistype something:

```
-bash: tcc: command not found
```

Know Your History, Because You're Going to Repeat It

One handy shell feature you should take advantage of is the history. Various history commands allow you to recall previously typed text at the command prompt. This is commonly done as you edit, compile, re-edit, and recompile your code.

For example, most of the time you're using this book you'll be cycling through three sets of commands. First comes the editing:

```
vim goodbye.c
```

Then comes the compiling:

```
gcc -lncurses goodbye.c
```

Then comes the running:

```
./a.out
```

I'll cover these steps in detail later, but for now recognize that these commands are to be repeated over and over: Edit, compile, run (or test); then re-edit, recompile, and test again. To assist you in that task, employ your shell's history function.

In the Bash shell, for example, use the up arrow key on your keyboard to recall a previous command. To recall the second previous command, press the up arrow key twice. I'm not intimate with the other shells, so if you use the C shell or Bourne shell, review your documentation for any history commands available with those shells.

Make a Place for Your Stuff

Please do be organized and build yourself a handy little directory into which you can save, compile, and test the various programs presented in this document.

For example, in my home directory, I have the following set up:

```
$HOME/prog/c/ncurses
```

`$HOME` is the home directory, the shell variable that represents your account's home directory for most UNIX shells that I've played with. It can also be abbreviated as `~/` in some shells.

Then I have a subdirectory called `PROG`, which contains all my programming junk and test files. `PROG` contains subdirectories for C language programs, Perl programs, shell scripts, and whatever else I'm dabbling in.

The `C` subdirectory contains C programs and directories.

Finally, the `NCURSES` directory is where I built all the sample files for this book.

You should consider a similar setup for your system, even if it's just something like `$HOME/ncurses`. As long as you can keep all the sample files around and be able to access them later, you'll be a happy camper.

If you want to create a `~/PROG/C/NCURSES` directory for your stuff, you can use the following command in your `home` directory:

```
mkdir -p prog/c/ncurses
```

The `-p` switch directs `mkdir` to build all parent directories to the final `NCurses` directory.

Using an Editor to Create an NCurses Program

There's no point in bothering with a fancy developer environment or IDE when you're programming NCurses. I think you'll be happier using the terminal window and a shell prompt, unless you've been totally corrupted by some IDE. Then you're on your own!

Picking an Editor

Since day one of UNIX, a text editor has been used to create code. That's what I recommend for this book. Any text editor will do, and most UNIX-like operating systems give you a smattering of editors to choose from:

- **ee.** The "easy editor" is a popular choice for many UNIX newcomers. No one will think any less of you for using ee, especially if you're using it with your C programming.
- **emacs.** This is the most popular choice, mostly because its commands are more word processor-like and you don't have to keep whacking the Escape key as you do in vi/vim.

- **vim.** This is my personal choice, simply because it's so damn raw and complex. As you get used to vim, though, it becomes a very powerful and handy tool. Plus it's common to all Unixes.

Whenever this book tells you to edit or create some source code, you'll use your favorite text editor to make it happen. (And please do create these programs in your NCurses directory, as covered in the previous section.)

If you don't know any editors, I recommend ee as the easiest. Otherwise, this book does not teach you how to use any text editor; I assume you'll figure that out on your own.

Creating Your First NCurses Program

Rather than just discuss all this stuff, why not get moving?

Use the `cd` command to change directories to the `NCURSES` directory you just created. You can confirm which directory you're using with the `pwd` command.

This is what I see on my screen:

```
/HOME/DANG/PROG/C/NCURSES
```

Your screen will probably show something different. The point is the same: You're in the `NCURSES` directory and ready to create some source code with your editor.

Source code is presented in this book as follows: First comes the filename, then the source code. To the left are line numbers for reference purposes only. Do not type the line numbers!

Use your editor to name (or create) the file; then input all the text *exactly* as shown in Listing 1-1.

Listing 1-1: GOODBYE.C

```
 1    #include <ncurses.h>
 2
 3    int main(void)
 4    {
 5        initscr();
 6        addstr(Goodbye, cruel C programming!);
 7
 8        endwin();
 9        return 0;
10    }
```

So if you're using vim, you would type:

```
vim goodbye.c
```

Then you would enter the text into the editor using your favorite, cryptic vim commands.

NOTE **Note that some compilers require there to be an extra blank line following the last line of code. This is not shown above or in any sample code in this document.**

When you're done entering text, double-check to ensure that you didn't miss anything.

Note that from now on it's assumed that whenever you see source code as shown here, you are to type it and name it according to the source code heading. And, naturally, you don't have to type *every* program, only those you want to experiment with.

Some Deviations

The next step in the programming process is compiling and linking, handled deftly by the common GCC command. But before compiling and linking, consider a few sidetracks, just to get you oriented if you're not used to programming in UNIX.

Use the `ls` command to view the contents of your NCURSES directory.

The `ls` command displays or lists the files in the directory, one of which should be `goodbye.c`. Confirm that.

```
~/prog/c/ncurses$  ls
goodbye.c
~/prog/c/ncurses$
```

You can also use the *long* variation on the `ls` command to see more details.

```
~/prog/c/ncurses$  ls -l
total 8
-rw-r--r--   1 dang  dang  113 dec  7 13:02 goodbye.c
~/prog/c/ncurses$
```

Now you can see permissions, owner, group, file size, and date information for the `GOODBYE.C` file — all of which help to confirm the file's existence.

Finally, you can view the file's contents with the `cat` command:

```
~/prog/c/ncurses$   cat goodbye.c
#include <ncurses.h>

int main(void)
{
        initscr();
```

```
        addstr(Goodbye, cruel C programming!);

        endwin();
        return 0;
}

~/prog/c/ncurses$
```

And there is the file yet again on the screen.

Typing `ls` and `cat` are not required steps in the program-creation process. I just like to remind you of their use here, which I liken to peering into the mail drop box twice just to confirm that your mail actually made it into the box and is not somehow stuck on the hinged lid.

Time to compile!

Know Thy Compiler

The standard C compiler in the UNIX environment is `gcc`, the GNU C compiler. Here is how it works in this book: You will see source code listed, such as the `goodbye.c` program. You will immediately know to type it and compile it.

To compile, you will type something at the shell prompt, perhaps like this:

```
gcc goodbye.c -lncurses
```

That's the `gcc` command, your compiler.

The first option is the name of the source code file, the text file you created. In this case, it's named `goodbye.c`. The single, lowercase `c` denotes a standard C source code file, not C++.

Finally comes `-lncurses`, which tells the compiler to `-l` "link in" the `NCurses` library. *This is very important*! `NCurses` is not just a header file; it's also a *library*. And you must link in the library to have those NCurses functions work.

Use this command:

```
gcc goodbye.c -lncurses
```

And you're compiled. Or not.

Linking NCurses or Curses?

On most systems I've visited, both the `CURSES` and `NCURSES` libraries are the same thing, meaning that if you link in `-lcurses` instead of `-lncurses`, the results are the same. The only advantage here is that typing `-lcurses` saves you a keystroke. Otherwise, I recommend using `-lncurses`.

What Does the gcc Command Do?

The `gcc` command either outputs a slew of error messages or shows you nothing.

When you get a slew of error messages, you must re-edit the source file and try to work out whatever bugs you can. The compiler is brutally honest, but it's also nice in that it does give you a line number to show you where (approximately) you screwed up.

When `gcc` does nothing, the source code is properly compiled and linked. This is what you want.

In this case, I've tricked you into typing sloppy code so that you'll see an error message. Something like:

```
goodbye.c:6: macro 'addstr' used with too many (2) args
```

One variation of the `gcc` compiler yielded even more information:

```
goodbye.c:6:45: macro "addstr" passed 2 arguments, but takes just 1
```

These error messages are just oozing with information:

- `goodbye.c` tells you which source code file is offensive.
- The `6` tells you that the error is either in line 6 or the previous line. In the second example, the `45` tells you which column in the line is offensive — very specific.
- Then the error message itself; something is apparently wrong with the call to the `addstr` macro. Must fix.

NOTE **If you didn't see the error message, you probably have been coding C for some time and just put the `addstr()` function's text in double quotes out of habit. Good for you!**

Re-editing Your Source Code

In programming you do more re-editing than editing. In this case, the error was on purpose so I could show you how the compiler displays an error message. The fix is easy: Just edit the `GOODBYE.C` source code file again.

Don't forget to use your shell's history (if available) to recall that editing command!

NOTE **Here's a tip: Familiarize yourself with the editor's command that instantly jumps to a specific line number. Most of your editing will actually be re-editing, where the compiler directs you to a specific line number. If you know the line-number-jumping command, you can get there quickly to fix your source code and try (again) to compile it:**

In vim, the line number skipping command is `nG`, where `n` is the line number and `G` is Shift+G. Thus, typing 6G will get you right to line 6.

The line should read:

```
addstr("Goodbye, cruel C programming!");
```

Then you should save the file to disk and re-compile it. But nothing happens. That's good! However....

Where Is the Program?

The program `gcc` creates is named `a.out`. It's a binary file, and its permissions are all properly set so that the operating system knows it's a program file and not a slice of Velveeta.

Use the `ls` command to confirm that `a.out` exists, if you like.

To run the program, you need to focus on the current directory: `./A.OUT`.

You can't just type `a.out`, because the operating system looks only to the search path for programs to run. So you must specifically direct tired old UNIX to look in the current directory — abbreviated by the **.** single dot — to run the program.

So `./` means "look in the current directory" and `A.OUT` means "run the file named `a.out`."

Of course, if you have the manual dexterity, you can always type a full pathname, something like:

```
~/prog/c/ncurses/a.out
```

This also runs the `a.out` program, but I believe you'll find typing `./A.OUT` a lot easier.

Nothing happens, not even an error. Again, there is a problem and you need to re-edit and recompile.

Fixing Stuff (Again)

Fixing stuff (again) in this case means that you forgot a key NCurses command. (Or more properly, fixing it again here means that I didn't specify a command on purpose simply to drive this point home.)

The problem? You didn't use the `refresh()` function, which is a common blunder in NCurses programming. Only by using `refresh()` is the NCurses "window" updated and any text written to the screen displayed. So, back to the editor!

Insert the `refresh()` function after the `addstr()` function on line 6. Your code should look like Listing 1-2, complete.

Listing 1-2: `goodbye.c`

```
1    #include <ncurses.h>
2
3    int main(void)
4    {
5        initscr();
6        addstr("Goodbye, cruel C programming!");
7        refresh();
8
9        endwin();
10       return 0;
11   }
```

Double-check your work.

Remember that you can use your shell's history to quickly recall those common commands: your editor, your compiler, and the `./a.out` command.

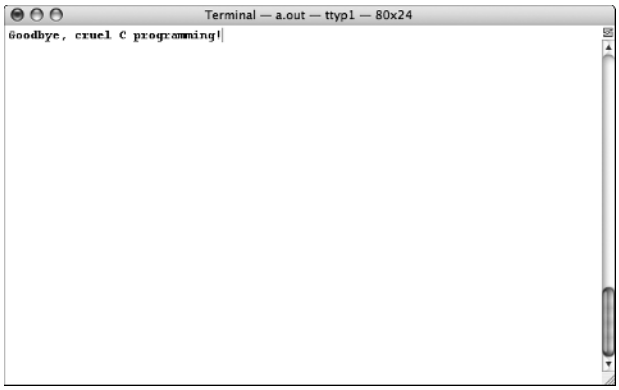

Figure 1-2: Output of the `GOODBYE.C` code.

Now it should work, and you'll see the string thrown up onto the screen via NCurses, as shown in Figure 1-2. Congratulations!

Don't Panic When You Still Don't See Anything!

Even with the `refresh()` function in the code, it's still possible that you won't see any program output. The problem isn't the program or even NCurses; it's your terminal.

Many terminals, such as xterm, support a feature known as *rmcup*. It restores the screen to what it looked like before a program was run. The situation also occurs with any full-screen terminal program, such as *man* or *less*; the program's text disappears after you quit the program, and the prompt "window" is restored.

Sadly, there is no handy way to switch off *rmcup* support from a terminal window. The `terminfo` file for the terminal needs to be recompiled to remove *rmcup* support, or a new `terminfo` file needs to be created in your home directory, one that lacks *rmcup* as an option.

The quick solution is to use the `getch()` function in your code. By inserting a line with `getch()` before the `endwin()` function, you can pause output and see what NCurses does before the program quits, as shown in Listing 1-3.

Listing 1-3: `goodbye.c`

```
 1    #include <ncurses.h>
 2
 3    int main(void)
 4    {
 5        initscr();
 6        addstr("Goodbye, cruel C programming!");
 7        refresh();
 8        getch();
 9
10        endwin();
11        return 0;
12    }
```

The new line 8 was added, allowing the program to pause, and for you to read the output.

Many of the program examples in this book use `getch()` to pause output. But some programs do not; be sure to use `getch()` in your code to see output, or modify your `terminfo` file to disable the *rmcup* feature.

NOTE **It might also help to be vocal about the *rmcup* feature for future releases of your operating system. While many folks may see *rmcup* as a handy thing, other users dislike it. The solution is to make the feature easy to disable. Let's hope that will be possible sooner than later.**

Do You Think a.out Is a Goofy Name?

Yes, `a.out` is a goofy name, but that's because the compiler doesn't know any better.

For running the myriad test programs in this book, using `a.out` will be a blessing. It won't take up as much disk space as individually compiling each program and creating separate silly little programs, plus it means you can instantly recall the `./a.out` command using your shell's history command.

But anyway, if you'd rather compile to a different output file, you need to specify the `–o` switch when you use `gcc`. It goes like this:

```
gcc goodbye.c -lncurses -o goodbye
```

`gcc` is still the compiler.

`goodbye.c` is the source code.

`-lncurses` directs the compiler to link in the *NCurses* library.

And finally, `-o goodbye` tells `gcc` to create the output file named `goodbye` as opposed to creating `a.out`.

Use the preceding command to accomplish this.

Do not forget the **./** prefix! Silly old UNIX needs to know where to find the file. So you must type `./GOODBYE` to run the program.

By the way, the output file doesn't have to be the same name as the source code file. You could use the following command if you like:

```
gcc goodbye.c -lncurses -o cloppyfeen
```

This creates the program file named `cloppyfeen` from the source code found in `goodbye.c`., so what you name the final program file can be anything you like.

All Done!

That pretty much does it for your whirlwind introduction to NCurses programming using the C language in the UNIX environment. This chapter has imparted the following knowledge, stuff that you'll need to carry with you throughout the remainder of this document:

General Info

Keep in mind that it's a good idea to keep your learning NCurses files in your special `NCURSES` directory. This is assumed.

Do remember those handy shell history commands. You'll be doing a lot of repetitious commands here, and pressing the up arrow key is a lot easier than retyping boring old UNIX commands.

And from now on, I will not be reminding you to specifically input, compile, and run the sample programs. There may be other, specific instructions given in the text, but whenever you see source code, it's assumed that you can type it in and run it if you want to learn more.

Handy Shell Commands to Know

`cat`	Displays a text file (source code) to the screen
`clear`	Clears the screen
`cp`	Copies a file
`ls -l`	Lists files in the long format
`ls`	Lists files
`mv`	Moves or renames a file
`rm`	Removes (deletes) a file

Source Code Tidbits

End the source code file with `.C` to show that it's a C language source code file. (Some editors, such as vim, may even recognize this and bless you with color-coded, in context contents as you edit.)

The `main()` function is an `int` and must return a value to the shell via either `return` or the `exit()` function.

If you use the `exit()` function, remember to include the `STDLIB.H` header file at the top of your source code.

If the program seems not to display anything, remember to add a `getch()` function before the `endwin()` function.

Compiling Tips

The compiler used in this book is `gcc`.

You must link in the NCurses library by using the `-lncurses` option to properly compile these programs.

The program file produced is always named `a.out`.

You must type `./a.out` to test run the program file.

You can use the `-o` compiler option to specify the name of the output file as something different from `a.out`.

The compiler command format is:

```
gcc filename.c -lncurses
```

You supply the `filename` according to the source code name given in this document.