Terminal Codes

In this lesson you will learn about how the terminal can be manipulated to display non-standard characters that direct it to change its appearance.

WE'LL COVER THE FOLLOWING

Although not directly related to bash, if you spend any time at a terminal, then it will pay off to understand how the terminal works with *non-standard characters*. You've already learned about the readline library and terminal options and how certain keystrokes can be 'caught' and handled in other ways before they get to the terminal. Here we look at how other non-standard characters can be handled in the bash shell.

Non-standard characters are characters like *tab*, *newline*, *carriage return*, and even the *end of file* characters. They don't form part of words, or necessarily print anything to the screen, but they are bytes interpreted by the shell and the terminal if they get that far.

> Note: The focus here is on ANSI-standard escape codes. Rarely, you might

> come across more complex escapes for specific terminal contexts, but this is beyond the scope of a 'practical' guide.

How Important is this Lesson?

This lesson is somewhat more advanced, and probably not essential to using bash at the start. It can also be hard to grasp, so if you're new to bash I recommend returning to it later if it's hard to understand the first time you read it.

However, knowledge of this area will catapult you to an elite of bash users that understand how terminals can be manipulated, and also enable you to understand how your prompt can be manipulated.

Non-Printable Characters

The terminal you use has what are described as *printable* characters and *non-printable* characters.

For example, typing a character like `a` (normally) results in the terminal adding an `a` to the screen. But there are other characters that tell the terminal to do different things that don't necessarily involve writing a character you'd recognise.

It's easy to forget this, but not everything that is sent to the computer is directly printed to the screen. The terminal *driver* takes what it is given (which is one or more bytes) and decides what to do with it. It might decide to print it (if it's a 'normal' character), or it might tell the computer to emit a beep, or it might change the colour of the terminal, or delete and go back a space, or it might send a message to the running program to tell it to exit.

When looking at non-printable characters, it's useful to be aware of a couple of utilities that help you understand what's going on. The first of these is a familiar one: `echo`.

Using `echo`

You're already familiar with echo, but it has some sometimes-useful flags you've not already used in this book:

```
echo -n 'JUST THESE CHARACTERS'
```

Type the above code into the terminal in this lesson.

Terminal

The `-n` flag tells echo not to print a newline.

```
echo -n $'JUST THESE CHARACTERS AND A NEWLINE\n'
```

Type the above code into the terminal in this lesson.

The `$` before the string makes us sure that bash will interpret `\n` as a newline on the screen, and won't just print out the characters `\n`. Try it without the `$` if you're unsure.

Here, it is the backslash character `\` that makes `echo` aware that a 'special' character is coming up. Being able to add a newline in this way means that you can send a newline to the terminal via the `echo` command, without confusing the command line by hitting the return key.

Other special characters include `\b` (for backspace), `\t` (for tab) and `\\` (to output a backslash):

Before you hit return, have a guess as to what this will output?

```
echo -n $'a\bb\bcde\bfg\b\b\b\n'
```

Type the above code into the terminal in this lesson.

If you guessed correctly, then well done! If you're struggling to understand what happened, note that a backspace does not delete the previous character, it just moves the cursor back a space.

You can also send a specific byte value to the terminal by specifying its hex value:

```
echo -n $'\x20\n'
```

Type the above code into the terminal in this lesson.

Think about that - you can use `echo` with these flags to control *exactly* what

gets sent to the screen. This is extremely valuable for debugging, or controlling what gets sent to the terminal.

It also bypasses the 'catching' of some characters you've seen from the previous readline lesson.

`CTRL-v` Escaping

Being able to output any binary value to the screen that we choose is useful, but what if we want to just output a 'special' character, and bypass the terminal's interpretation via terminal options or the readline library?

For example, if I hit `TAB` in my terminal it would normally *not* show a tab character (or move my cursor along a few spaces), as the readline library uses the `TAB` key (if hit twice in a row) to auto-complete any text we have not finished.

But if I'm typing something like:

```
echo 'I want a tab here:>X<a tab'
```

Type the above code into the terminal in this lesson.

How do I get a 'real' tab where the `X` is?

This is one way: instead of the `X`, you type `\C-v` and then `\C-i`.

If you type this in a terminal at the bash prompt, the cursor will tab across the screen in the way you might have previously expected.

If you look at the output of `stty -a` again (as you did in the previous readline lesson) then you will that `^V` is bound to the `lnext` action. I believe (but can't confirm) that this stands for 'next character literal'.

```
discard dsusp   eof     eol     eol2    erase   intr    kill    lnext
^O      ^Y      ^D      <undef> <undef> ^?      ^C      ^U      ^V
min     quit    reprint start   status  stop    susp    time    werase
1       ^\      ^R      ^Q      ^T      ^S      ^Z      0       ^W
```

There are multiple characters represented in this way. You've just seen tab (technically a *vertical tab*) represented as `^I`, and backspace represented as `^H`.

If you want to see a table of the possible shortcuts that may be seen, then type `man ascii` into the terminal above. The table begins like this:

```
020   16    10    DLE (data link escape)      120   80    50    P
-------------------------------------------------------------------
000   0     00    NUL '\0' (null character)   100   64    40    @
001   1     01    SOH (start of heading)      101   65    41    A
002   2     02    STX (start of text)         102   66    42    B
003   3     03    ETX (end of text)           103   67    43    C
004   4     04    EOT (end of transmission)   104   68    44    D
005   5     05    ENQ (enquiry)               105   69    45    E
006   6     06    ACK (acknowledge)           106   70    46    F
007   7     07    BEL '\a' (bell)             107   71    47    G
010   8     08    BS  '\b' (backspace)        110   72    48    H
011   9     09    HT  '\t' (horizontal tab)   111   73    49    I
012   10    0A    LF  '\n' (new line)         112   74    4A    J
013   11    0B    VT  '\v' (vertical tab)     113   75    4B    K
014   12    0C    FF  '\f' (form feed)        114   76    4C    L
015   13    0D    CR  '\r' (carriage ret)     115   77    4D    M
```

How would you input this, therefore, and what will it output?

```
echo abcc^Hdefg
```

As the `echo` is processed, it should output:

```
abcdefg
```

because the second `c` is deleted by the `^H` you input by hitting `^V` (or `\C-v`) followed by `^H` (or `\C-h`).

Carriage Returns vs Line Feeds

The most commonly-seen non-printable character is the *carriage return*.

Carriage returns and line feeds cause much confusion, but it doesn't take long to understand the difference and (more importantly) why they are different.

If you think about an old-fashioned typewrite or printer that moves along punching out characters to a page, at some point it has to be told: 'go back to the beginning of the line'. Then, once at the beginning of the line, it has to be told: 'feed the paper up one line so I can start writing my new line'.

A *carriage return* is, as the word 'return' suggests, 'returns' the cursor to the start of the line. It's represented by the character `r` for return. The *line feed*, again as the name suggests, feeds the line up. In a modern terminal, this just means 'move the cursor down'.

So far, so clear and simple to learn. But, Linux does things differently! In Linux, `\n` is sufficient to do both. In Windows, you need both the `\r` and `\n` characters to represent a new line.

This means that files can 'look funny' in Linux terminals with these weird `^M` characters showing at the end of each line. To confuse things even more, some programs automatically handle the difference for you and hide it from you.

So what will this output?

```
echo $'Bad magazine\rMad'
```

The difficulty of understanding how bytes are turned into the output you see on the terminal is why it's important to have a way to see what the actual bytes in a file are. Here a very useful tool comes in: `hexdump`.

Hexdump

Run this:

```
echo $'Bad magazine\rMad' | hexdump
echo $'Bad magazine\rMad' | hexdump -c
```

Type the above code into the terminal in this lesson.

Hexdump prints out the characters received in standard input as hex digits. 16 characters are printed per line, and on the left is displayed the count (also in hex) of the number of bytes processed up to that line.

The `-c` flag prints out the contents as characters (including the control ones with appropriate backslashes in front, eg `\n`), whereas leaving it out just displays the hex values.

It's a great way to see what is *really* going on with text or any stream of output of bytes.

If you go back to the first example in this section:

```
echo 'JUST THESE CHARACTERS' | hexdump -c
echo -n 'JUST THESE CHARACTERS' | hexdump -c
```

Type the above code into the terminal in this lesson.

You can figure out for yourself the difference between using the `-n` flag in `echo` and not using it.

Terminal Escape Codes

A terminal escape code is a defined sequence of bytes that, if sent to the terminal, will perform a specific action.

Run this:

```
echo $'\033[?47h'
echo $'\033[?47l'
```

Type the above code into the terminal in this lesson.

The first line 'saves' the screen and the second restores it.

The ANSI codes always start with the `ESC` character (`033` in octal) and left bracket character: in hex `1B`, then `5b`. So you could rewrite the above as:

```
echo $'\x1b\x5b?47h'
echo $'\x1b\x5b?47l'
```

Type the above code into the terminal in this lesson.

These `ESC` and `[` characters are then followed by specific sequences which can change the colour of the screen, the background text, the text itself, set the screen width, or even re-map keyboard keys.

Type this out and see if you can figure out what it's doing as you go:

```
ansi-test() {
for a in 0 1 4 5 7
    do
    echo "a=$a "
    for (( f=0; f<=9; f++ ))
    do
```

```
        for (( b=0; b<=9; b++ ))
        do
            echo -ne "\\033[${a};3${f};4${b}m"
            echo -ne "\\\\\\\\\033[${a};3${f};4${b}m"
            echo -ne "\\033[0m "
        done
        echo
    done
    echo
done
echo
}
```

Type the above code into the terminal in this lesson.

The output shows you what all the ANSI terminal escape codes are and you can see what they do in the terminal.

Sometimes when you `cat` a binary file, (or `/dev/random`, which outputs random bytes) the contents when output to a terminal can cause the terminal to appear to 'go haywire'. This is because these escape codes are accidentally triggered by the sequences of bytes that happen to exist in these files.

1 What will this output?

```
echo -n $'12345\r54321'
```

What You Learned

- What *terminal codes* are
- What *printable* and *non-printable* characters are
- How to output any arbitrary bytes
- How to input a literal character using `CTRL-v`
- The difference between `\n` and `\r\n`
- What *terminal escape codes* are

What Next?

Building on this knowledge, next you will learn how to set up your prompt so that it can show you (and even do) useful things.

Exercises

1) Research and `echo` all of `echo`'s escape sequences. Play with them and figure out what they do.

2) Research and `echo` ten terminal escape sequences.

3) Look up all the `CTRL-v` escape sequences and experiment with them.

4) Research the command `tput`, figure out what it does and rewrite some of the above commands using it.

5) Re-map your keyboard so it outputs the wrong characters using escape codes.