```
When the only hammer you have is C++, the whole world looks like a thumb.
 -- Keith Hodges
```

Deep Thoughts

At this point, we're getting pretty close to what I consider the upper limit of basic shell scripting; there are still a few areas I'd like to cover, but most of the issues are getting rather involved. A good example is the 'tput' command that I'll be discussing this month: in order to really understand what's going on, as opposed to just using it, you'd need to learn all about "termcap/terminfo" controversy - a deep, involved, **ugly** issue (for those who really want to know, just search the Net and read the "Keyboard-and-Console-HOWTO".) I'll try to make sense despite the confusion, but be warned.

Affunctionately Yours

The concept of functions is not a difficult one, but is certainly very useful: they are simply blocks of code that you can execute under a single label - essentially, a way to save you a lot of typing when repeating previous work. Unlike a script, they do not spawn a new subshell but execute within the current one. Functions can be used within a script, or as stand-alone chunks of code to be executed in the shell.

Let's see how a function works in a shell script:

```
#!/bin/bash
#
# "venice beach" - translates English to beach-bunny

# Makes everything, like, totally rad, dude!
function kewl ()
{
         [ -z "$1" ] && {
                  # Exit with an error message if no argument has been
                  # supplied.
                  echo "That was bogus, dude."
                  return 1
         }

         echo "Like, I'm thinkin', dude, gimme a minute..."
         sleep 1

         # While the function runs, positional parameters ($1, etc.)
         # refer to those given the function - not the shell script.
         echo $*', dude!'
}

# Process all the command-line arguments
kewl $*
```

This, umm, incredibly important script should print the "I'm thinkin'..." line followed by a thoroughly mangled list of parameters:

```
Odin:~$ venice_beach Right on
Like, I'm thinkin', dude, gimme a minute...
Right on, dude!
Odin:~$ venice_beach Rad.
Like, I'm thinkin', dude, gimme a minute...
Rad, dude!
Odin:~$ venice_beach Dude!
```

```
Like, I'm thinkin', dude, gimme a minute...
Dude, dude!
```

Functions may also be loaded into the environment, and invoked just like shell scripts; we'll talk about sourcing functions later on. For those of you who use Midnight Commander, check out the "mc ()" function described in their man page - it's a very useful one, and is initially loaded from ".bash_profile".

Important item: functions are *created* as "`function pour_the_beer () { ... }`" or "`pour_the_beer () { ... }`" (the keyword is optional), but they are *invoked* as "pour_the_beer" (no parentheses). Also, remember not to use an "exit" statement in a function: since you're running the code in the current shell, this will cause you to exit your current shell or terminal! Exiting a shell script this way can produce some very ugly results, like a 'hung' console that has to be killed from another VT (yep, I've experimented). The statement that will terminate a function without killing the shell is "return".

Single, Free, and Easy

Everything we've discussed in this series so far has a common underlying assumption: that the script you're writing is going to be saved and re-used. For most scripts, that's what you'd want - but what if you have a situation where you need the *structure* of a script, but you're only going to use it once (i.e., don't need or want to create a file)? The answer is - Just Type It:

```
Odin:~$ (
> [ "$blood_caffeine_concentration" -lt 5ppm ] && {
> echo $LOW_CAFFEINE_ERROR
> brew ttyS2      # Enable coffeepot via /dev/ttyS2
> while [ "$coffee_cup" != "full" ]
> do
> echo "Drip..."
> sleep 1
> done
>
> while [ "$coffee_cup" != "empty" ]
> do
> sip_slowly      # Coffee ingestion binary, from coffee_1.6.12-4.tgz
> echo "Slurp..."
> done
> }
>
> echo "Aaaahhh!"
> echo
> )
Coffee Not Found: Operator Halted!
Drip...
Drip...
Drip...
Slurp...
Slurp...
Slurp...
Aaaahhh!

Odin:~$
```

Typing a '(' character tells Bash that you'd like to spawn a subshell and execute, within that subshell, the code that follows - and this is what a shell script does. The ending character, ')', obviously tells the subshell to 'close and execute'.

So, why would we want to do this? Well, there are many times when you might want to execute a function as an atomic unit - a good example might be restarting an SSH server on a remote machine.

```
# BAD idea!
ssh root@remotehost '/etc/init.d/sshd stop; /etc/init.d/sshd start'
```

Think about it: the first command shuts down the server, the second one restarts it... OOOPS! As soon as you shut it down, you lost your connection to that machine - worse yet, now that the server is down, you can't even reconnect! The right answer, of course, is to run it in a subshell:

```
ssh root@remotehost '(/etc/init.d/sshd stop; /etc/init.d/sshd start)'
```

Even if you're not logged in, the subshell will finish executing.

Explicit subshells are also useful for delayed/timed operations:

```
# Shows 10 seconds worth of incoming system messages, then exits
(sleep 10; pkill -n tail) & tail -f /var/log/messages
```

Of course, something like a simple loop or a single 'if' statement doesn't require any of that; it can simply be typed in and run from the command line:

```
Odin:~$ for fname in *.c
> do
> echo $fname
> cc $fname -o $(basename $fname .c)
> done
```

"bash" is smart enough to recognize a multi-part command of this type - a handy sort of thing when you have more than a line's worth of syntax to type (not an uncommon situation in a 'for' or a 'while' statement). By the way, a cute thing happens when you hit the up-arrow to repeat the last command: "bash" will reproduce everything as a single line - with the appropriate semi-colons added. Clever, those GNU people...

No shebang ("#!/bin/bash") is necessary for a one-time script, as there would be at the start of a script file. You know that you're executing this in "bash" (at least I *hope* you're running "bash" while writing and testing a "bash" script...), whereas with a script file you can never be sure: the user's choice of shell is an arbitrary variable, so the shebang is necessary to make sure that the script uses the correct interpreter.

The Best Laid Plans of Mice and Men

In order to write good shell scripts, you have to learn good programming. Simply knowing the ins and outs of the commands that "bash" will accept is far from all there is - the first step of problem *resolution* is problem *definition*, and defining exactly what needs to be done can be far more challenging than writing the actual script.

One of the first scripts I ever wrote, "bkgr" (a random background selector for X), had a problem - I'd call it a "race condition", but that means something different in Unix terminology - that took a long time and a large number of rewrites to resolve. "bkgr" is executed as part of my ".xinitrc":

```
...
# start some nice programs
bkgr &
rxvt-xterm -geometry 78x28+0+26 -tn xterm -fn 10x20 -iconic &
coolicon &
icewm
```

OK, going by the book - I background all the processes except the last one, "icewm" (this way, the window manager keeps X "up", and exiting it kills the server). Here was the problem: "bkgr" would run and "paint" my background image on the root window; fine, so far. Then, "icewm" runs - and paints a greenish-gray background over it (I found out later that there's a way do disable that behavior, but the problem it presented at the time was interesting enough to be worth examining.)

What to do? I can't put "bkgr" after "icewm" - the WM has to be last. How about a delay after "bkgr", say 3 seconds... oh, that won't work: it would simply delay the "icewm" start by 3 seconds. OK, how about *this* (in "bkgr"):

```
...
while [ -z "$(ps ax|grep icewm)" ] # Check via 'ps' if "icewm" is up
do
         sleep 1 # If not, wait, then loop
done
...
```

That should work, since it'll delay the actual "root window painting" until after "icewm" is up!

It didn't work, for *three* major reasons.

Reason #1: try the above "ps ax|grep" line from your command line, for any process that you have running; e.g., type

```
ps ax|grep init
```

In fact, try it several times. What you will get, randomly, is either one or two lines: just "init", or "init" *and* the "grep init" as well, where "ps" manages to catch the line that you're currently executing!

Reason #2: "icewm" starts, takes a second or so to load, and *then* paints the root window. Worse yet, that initial delay varies - when you start X for the first time after booting, it takes significantly longer than subsequent restarts. "So," you say, "make the delay in the loop a bit longer!" That doesn't work either - I've got two machines, an old laptop and a desktop, and the laptop is horribly slow by comparison; you can't "size" a delay to one machine and have it work on both... and in my not-so-humble opinion, a script should be universal - you shouldn't have to "adjust" it for a given machine. At the very least, that kind of tuning should be minimized, and preferably eliminated completely.

One of the things that also caused trouble at this point is that some of my pics are pretty large - e.g., my photos from the Kennedy Space Center - and take several seconds to load. The overall effect was to allow large pics to work with "bkgr", whereas the smaller ones got overpainted - and trying to stretch the delay resulted in a significant built-in slowdown in the X startup process, an undesirable side effect.

Reason #3: "bkgr" was supposed to be a random background selector as well as a startup background selector - meaning that if I didn't like the original background, I'd just run it again to get another one. A built-in delay any longer than a second or so, given that a pic takes time to paint anyway, was not acceptable.

What a mess. What was needed was a conditional delay that would keep running as long as "icewm" wasn't up, then a fixed delay that would cover the interval between the "icewm" startup and the "root window painting". The first thing I tried was creating a reliable 'detector' for "icewm":

```
...
delay=0
X="$(ps ax)"

while [ $(echo $X|grep -c icewm) -lt 1 ]
do
[ $delay -eq 0 ] && (delay=1; sleep 3)
[ $delay -eq 1 ] && sleep 1
X="$(ps ax)"
done
...
```

'$X' gets set to the value of "$(ps ax)", a long string listing all the running processes which we check for the presence of "icewm" as the loop condition. The thing that makes all the difference here is that "ps ax" and "grep" are not running at the same time: one runs inside (and just before) the loop, the other is done as part of the loop test. This registers a count of only one "icewm" if it is running, and none if it is not. Unfortunately, due to the finicky timing - specifically the difference in the delays between an initial X startup and repeated ones - this wasn't quite good enough. Lots of experimentation later, I came up with a version that worked - a rather crude hack, but one that finally reflected that I understood what was going on... only to be supplanted a week or two later by a new release of "icewm" that fixed/bypassed all those problems. **And** I found out about 'pgrep', which eliminates that 'multiple line' problem when checking the process table. Life was good. :)

There are a number of programming errors to watch out for: "race conditions" (a security concern, not just a time conflict), the 'banana problem', the 'fencepost' or 'Obi-Wan' error... (Yes, they do have interesting names; a story behind each one.) Reading up on a bit of programming theory would benefit anyone who's learning to write shell scripts; if nothing else, you won't be repeating someone else's mistakes. My favorite reference is an ancient "C" manual, long out of print, but there are many fine reference texts available on the net; take a peek. There are standard algorithms that solve standard programming problems; these methods are language-independent, and are very good gadgets to have in your mental toolbox.

Coloring Fun With Dick and Jane

One of the things that I used to do, way back when in the days of BBSs and the ASCII art that went with them, is create flashy opening screens that moved and bleeped and blinked and did all sorts of things. This was accomplished without any graphics programming or anything more complicated than ASCII codes and ANSI escape sequences - they could get complicated enough, thank you very much - since all of this ran on pure text terminals. (Incidentally, if you want to do that kind of thing, Linux **rocks**. Thanks to the absolutely stunning work done by Jan Hubicka and friends on "aalib" - if you have not seen "bb", the demo program for "aalib", you're missing out on a serious acid trip - it has outstripped everything even the fanciest ASCII artist could come up with and then some. As an example, they have "Quake" and fractal generators running on text-only terminals; hard to believe, but true - and beautifully done.)

What does this have to do with us, since we're not doing any "aalib"-based programming? Well, there are times when you want to create a nice-looking menu, say one you'll be using every day - and if you're working with text, you'll need some specialized tools:

1) Cursor manipulation. The ability to position the cursor is a must; being able to turn it on and off, and saving and restoring the position are nice to have.

2) Text attribute control. Bold, underline, blinking, reverse - these are all useful in menu creation.

3) Color. Let's face it: plain old B&W gets boring after a bit, and even something as simple as a text menu can benefit from a touch of spiffing up.

So, let's start with a simple menu:

```
#!/bin/bash
#
# "ho-hum" - a text-mode menu

clear

while [ 1 ] # Loop 'forever'
do
         # We're going to do some 'display formatting' to lay out the text;
         # a 'here-document', using "cat", will do the job for us.

         cat <<-!

         M A I N   M E N U

         1. Business auto policies
         2. Private auto policies
         3. Claims
         4. Accounting
         5. Quit
         !

         echo -n " Enter your choice: "

         read choice
         case $choice in
                 1|B|b) bizpol ;;
                 2|P|p) perspol ;;
                 3|C|c) claims ;;
                 4|A|a) acct ;;
                 5|Q|q) clear; exit ;;
                 *) echo; echo "\"$choice\" is not a valid option."; sleep 2 ;;
         esac

         sleep 1
         clear
done
```

If you download the file containing this script and run it, you'll realize why the insurance business is considered deadly dull. Erm, well, *one* of the reasons, I guess. Bo-o-ring ([grin] apologies to one of my former employers, but it's true...) Not that there's a tremendous amount of excitement to be had out of a text menu - but surely there's something we can do to make things just a tad brighter!

```
#!/bin/bash
#
```

```
# "jazz it up" - an improved text-mode menu

tput civis        # Turn off the cursor

while [ 1 ]
do
         echo -e '\E[44;38m' # Set colors: bg=blue, fg=white
         clear    # Redraw the screen with the above color set
         echo -e '\E[41;38m' # bg=red

         echo -ne '\E[45;38m' # bg=magenta
         tput cup 8 25 ; echo -n  "    M A I N     M E N U    "
         echo -e '\E[41;38m' # bg=red

         tput cup 10 25 ; echo -n " 1. Business auto policies "
         tput cup 11 25 ; echo -n " 2. Private auto policies  "
         tput cup 12 25 ; echo -n " 3. Claims                 "
         tput cup 13 25 ; echo -n " 4. Accounting             "
         tput cup 14 25 ; echo -n " 5. Quit                   "

         echo -ne '\E[44;38m' # bg=blue
         tput cup 16 28 ; echo -n " Enter your choice: "
         tput cup 16 48

         read choice
         tput cup 18 30

         case "$choice" in
                 1|B|b) bizpol ;;
                 2|P|p) perspol ;;
                 3|C|c) claims ;;
                 4|A|a) acct ;;
                 5|Q|q) tput sgr0; clear; exit ;;
                 *) tput cup 18 26; echo "\"$choice\" is not a valid option."; sleep 2 ;;
         esac
done
```

This is NOT, by any means, The Greatest Menu Ever Written - but if you run it in a modern terminal (i.e., xterm or rxvt), it will give you an idea of some basic layout and color capabilities. Note that the colors may not work exactly right in some odd versions of xterm, depending on your hardware and your "terminfo" version - I did this as a quick slapdash job to illustrate the capabilities of "tput" and "echo -e". These things can be made portable - "tput" variables are common to almost everything, and color values can be set based on the value of '$TERM' - but this script falls short of that. These codes, by the way, are basically the same for DOS, Linux, etc., text terminals - they're dependent on hardware/firmware rather than the software we're running. Xterms, as always, are a different breed...

So, what's this "tput" and "echo -e" stuff? Well, in order to 'speak' directly to our terminal, i.e., give it commands that will be used to modify the terminal characteristics, we need a method of sending control codes. The difference between these two methods is that while "echo -e" accepts "raw" escape codes (like '\e[H\e[2J' - same thing as <esc>H<esc>2J), "tput" calls them as 'capabilities' - i.e., "tput clear" does the same thing as "echo -e" with the above code. 'tput' is also term-independent: it uses the codes in the terminfo database for the current term-type. The problem with 'tput' is that most of the codes for it are as impenetrable as the escape codes that they replace: things like 'civis' ("make cursor invisible"), 'cup' ("move cursor to x y"), and 'smso' ("start standout mode") are just as bad as memorizing the codes themselves! Worse yet, I've never found a reference that lists them all... well, just remember that the two methods are basically interchangeable, and you'll be able to use whatever is available. The "infocmp" command will list the capabilities and their equivalent codes for a given terminal type; when run without any parameters, it returns the set for the current terminal.

Colors and attributes for an ISO6429 (ANSI-compliant) terminal, i.e., a typical text terminal, can be found in the "ls" man page, in the "DISPLAY COLORIZATION" section; xterms, on the other hand, vary

so much in their interpretation of exactly what a color code means, that you basically have to "try it and see":

```
#!/bin/bash
#
# "colsel" - a term color selector

# Restore original term settings on exit
trap 'reset' 0

# Cycle the background color set
for n in `seq 40 47`
do
         # Cycle the foreground color set
        for m in `seq 30 37`
                do
                echo -en "\E[$m;${n}m"
                  # Redraw screen with new color set
                clear
                  echo -n "The selected colors are "
                  # Set the colors to black and white for readability
                echo -en "\E[37;40m"
                  # Show the settings
                echo $n $m
                sleep 1
        done
done
```

This little script will run through the gamut of colors of which your terminal is capable. Just remember the number combos that appeal to you, and use them in your "echo -e '\E[<bg>;<fg>m'" statements.

Note that the positions of the numbers within the statement don't matter; also note that some combinations will make your text into unreadable gibberish ("12" seems to do that on most xterms). Don't let it bother you; just type "reset" or "tput sgr0" and hit "Enter".

Wrapping it Up

Hmm, I seem to have made it through all of the above without *too* much pain or suffering; amazing. :) Yes, some of the areas of Linux still have a ways to go... but that's one of the really exciting things about it: they *are* changing and going places. Given the amazing diversity of projects people are working on, I wouldn't be surprised to see someone come up with an elegant solution to the color code/attribute mess - or come up with something that eliminates the whole problem.

Next month, we'll cover things like sourcing functions (pretty exciting stuff - reusable code!), and some really nifty goodies like "eval" and "trap". Until then -

Happy Linuxing to all!

Linux Quote of the Month

"The words 'community' and 'communication' have the same root. Wherever
you put a communications network, you put a community as well. And whenever
you **take away** that network - confiscate it, outlaw it, crash it,
raise its price beyond affordability - then you hurt that community.

```
Communities will fight to defend themselves. People will fight harder
and more bitterly to defend their communities, than they will fight to
defend their own individual selves."
 -- Bruce Sterling, "Hacker Crackdown"
```

References The "man" pages for 'bash', 'builtins', 'tput', 'infocmp', 'startx'

"Introduction to Shell Scripting - The Basics", LG #53
"Introduction to Shell Scripting", LG #54
"Introduction to Shell Scripting", LG #55
"Introduction to Shell Scripting", LG #56
"Introduction to Shell Scripting", LG #57

Ben is the Editor-in-Chief for Linux Gazette and a member of The Answer Gang.

Ben was born in Moscow, Russia in 1962. He became interested in electricity at the tender age of six, promptly demonstrated it by sticking a fork into a socket and starting a fire, and has been falling down technological mineshafts ever since. He has been working with computers since the Elder Days, when they had to be built by soldering parts onto printed circuit boards and programs had to fit into 4k of memory. He would gladly pay good money to any psychologist who can cure him of the recurrent nightmares.

His subsequent experiences include creating software in nearly a dozen languages, network and database maintenance during the approach of a hurricane, and writing articles for publications ranging from sailing magazines to technological journals. After a seven-year Atlantic/Caribbean cruise under sail and passages up and down the East coast of the US, he is currently anchored in St. Augustine, Florida. He works as a technical instructor for Sun Microsystems and a private Open Source consultant/Web developer. His current set of hobbies includes flying, yoga, martial arts, motorcycles, writing, and Roman history; his Palm Pilot is crammed full of alarms, many of which contain exclamation points.

He has been working with Linux since 1997, and credits it with his complete loss of interest in waging nuclear warfare on parts of the Pacific Northwest.