Work the Shell

Calling All Functions, and Some Math Too!

Dave Taylor

Abstract

Shuffle off to the next step in creating a shell script Blackjack game.

If you've been keeping track of my column, you'll know that we're building a Blackjack game as a shell script. Why? Because most shell scripts are far too boring to study without nodding off, so considering all the nuances of shell script programming within the context of a game just sounds more fun!

My last column talked about how to shuffle a deck of cards, as portrayed in a program with a simple array of 52 values, 1-52. There are some interesting nuances to the shuffle problem. Let's start there. Then we'll look at how to turn an arbitrary 1-52 value into a familiar rank and suite from a deck of cards.

The Shuffle Function

If you've been writing shell scripts since the dawn of UNIX, you might not have realized that modern shells now support functions and procedures, just like a "real" programming language. For any block of code that you plan on executing more than once, it's the way to go.

Here's the shuffle code, written as a shell function:

```bash
function shuffleDeck
{
  count=1
  while [ $count -le 52 ]
  do
    pickCard
    newdeck[$count]=$picked
    count=$(( $count + 1 ))
  done
}
```

This builds an array called newdeck, which is actually the shuffled deck (in the last column we showed deck, which was an array of cards in linear order), and it uses some basic shell math with the $(( )) notation to save spawning a subshell to invoke expr for each increment of the count variable.

I said that shell scripts are robust programming environments and that might be a tiny bit of hyperbole, really. Eagle-eyed readers will notice that the pickCard function returns its value by setting a global variable, picked, which isn't really optimal programming strategy. But it works, and pragmatism is an important part of any good approach to software development, isn't it?

The full pickCard function is key to making this work, but it is too long to include here, so grab it from the LJ FTP site for your reading pleasure (ftp://ftp.ssc.com/pub/lj/listings/8774.tgz).

With the shuffleDeck function written and an initializeDeck function, as shown here:
function initializeDeck
{
  card=1 while [ $card -le 52 ] do
    deck[$card]=$card card=$(( $card + 1 ))
  done
}

it's easy to do the rudiments of shuffling the deck and dealing out two cards for the player and two for the dealer:

initializeDeck shuffleDeck

echo "** Player's hand: ${newdeck[1]}, ${newdeck[3]}"
echo "** Dealer's hand: ${newdeck[2]}, ${newdeck[4]}"

Let's run this and see what kind of results we get:

$ ./blackjack.sh
** Player's hand: 22, 49
** Dealer's hand: 11, 8

$ ./blackjack.sh
** Player's hand: 19, 32
** Dealer's hand: 49, 10

$ ./blackjack.sh
** Player's hand: 44, 23
** Dealer's hand: 46, 11

Displaying cards as a numeric value from 1-52 is not the most friendly, so let's turn our attention to the display of the card values in the familiar rank and suite of a traditional deck of playing cards.

** Math Games to Identify Rank and Suite **

A deck of cards is composed of 52 cards, split evenly into four suites of 13 cards. The order of the suites doesn't matter (in Blackjack, at least), but the rank does. Indeed, the goal of the game is have a summary rank value of 21 points without going any higher.

The rank of a card is the remainder of the numeric card value divided by 13. In math terms, this is called the modulus and can be computed thusly:

\[ \text{rank} = \text{cardvalue} \mod 13 \]

Getting the suite should be simple too; it's the other half of the division. In other words, if the card value is 17, then \( 17/13 = 1 \), meaning it's suite #1, and \( 17 \mod 13 = 4 \). Since we'd like to have our suites in the range of 1-4, rather than 0-3, however, we'll need to add one to the equation. Further, the 13th of each card is the same suite as the earlier 12, so we'll also have to subtract one before we do the division (card #13 would be suite #1 if we just calculated 13/13, but if we calculate 12/13 for that, we'll correctly identify it as part of suite #0).

This is very confusing, so here's the equation instead:

\[ \text{suite} = (( ( \text{card} - 1) / 13 ) + 1) \]

Much clearer, right? Seriously, you can experimentally verify that this works correctly. The important edge cases are value=1, value=12, value=13 and value=14. If you can get those right, you're good for all values in
the deck.

Once we've identified the rank and suite of a card, we just have to do a bit of fancy footwork to turn numbers into words:

```bash
case $suite in
  1 ) suite="Hearts"  ;;
  2 ) suite="Clubs"    ;;
  3 ) suite="Spades"  ;;
  4 ) suite="Diamonds" ;;
  * ) echo "Bad suite value: $suite"; exit 1
esac
```

and:

```bash
case $rank in
  0 ) rank="King"    ;;
  1 ) rank="Ace"     ;;
  11) rank="Jack"    ;;
  12) rank="Queen"  ;;
esac
```

Put these together in a function called showCard (which returns "$cardname" as the calculated rank and suite), and we can now clean up a bit:

```bash
initializeDeck shuffleDeck
echo -n "** Player's hand: 
    showCard ${newdeck[1]} ; echo -n "$cardname, 
    showCard ${newdeck[3]} ; echo "$cardname"
echo -n "** Dealer's hand: 
    showCard ${newdeck[2]} ; echo -n "$cardname, 
    showCard ${newdeck[4]} ; echo "$cardname"
```

Now we can start to see the game come together! Consider:

```
$ ./blackjack.sh
** Player's hand: 8 of Clubs, 3 of Diamonds
** Dealer's hand: King of Spades, 3 of Spades
$ ./blackjack.sh
** Player's hand: 2 of Spades, 4 of Spades
** Dealer's hand: 10 of Spades, 4 of Hearts
```

Let's stop here this month as that's a lot of code to dig through already. I invite you to pop over to the *LJ* FTP site to grab all the source code so far, so you can experiment with this script yourself too.

Next month, we'll start looking at the game logic itself, but for now, Vegas beckons for a big tradeshow and, well, I can write it off as research for *Linux Journal*, can't I?