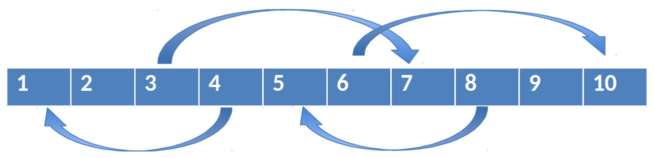
Example to show use of control structures.
Simplified snakes and ladders … to 10 not 100

If divisible by 3, take to the tree … 4 steps up the ladder!



If you overshoot,
Just bounce back,
By the same

If divisible by 4, go towards the floor… slide 3 steps down the snake!

Something fairly familiar which shows use of :-

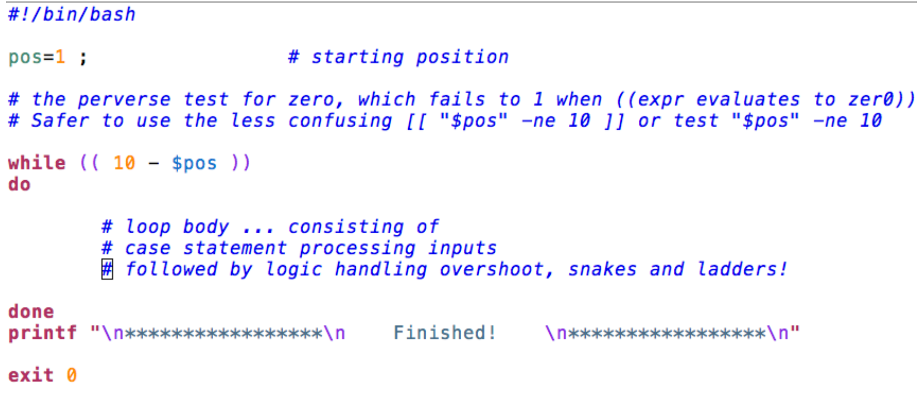
- If … test for jumps
- Case … processing throw of the die
- While … you're still in the game!

Snakes and ladders pseudocode algorithm… to 10

```
Starting position (pos =0)

// while still in the game…
While not at end position (10 for ease, 100 for real)

// throw a die & make a move
    throw a die  from [1-6] or 999 (emerge-ncy!?) to exit
    case
        [1-6] valid input, move by throw pos += throw
        999 break out of loop and finish
        otherwise : invalid input : ignore & loop again
    endcase
// bounceback by overshoot
    if pos exceeds end (overshoots endlimit)
    then retreat from end by excess (bounceback from end by overshoot)
    endif
// if divisible by 3, then take to the tree by 4…take 4 steps up the ladder!**
    if (pos==3) or (pos==6)
    then jump forward (via ladder) by 4
    endif
// if divisible by 4, hit the floor by 3…slide 3 down the snake!**
    if pos is a multiple of 4 (evenly divisible with remainder =0)
    then jump back (via snake) by 3
    endif

Endwhile
```

** to avoid confusion, the ladder up & slide down increments avoid repeated steps/slides – but Exercise at end overrides this.

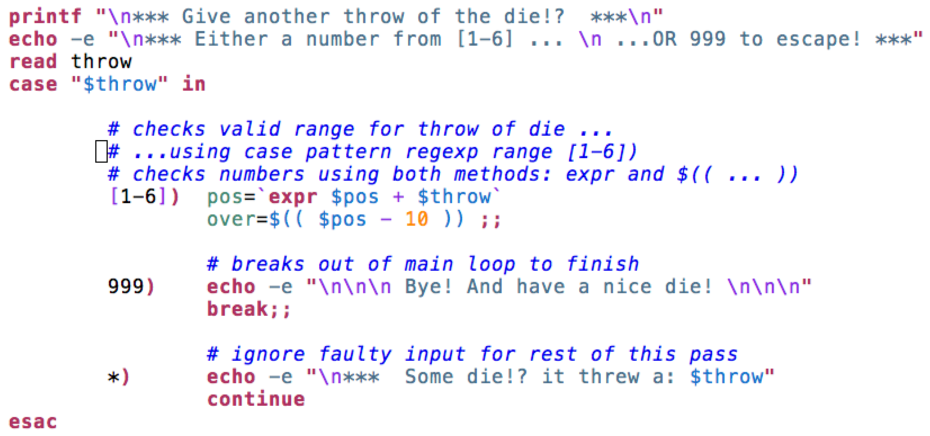
The main loop – expanded for clarity!



```
#!/bin/bash

pos=1 ;                    # starting position

# the perverse test for zero, which fails to 1 when ((expr evaluates to zer0))
# Safer to use the less confusing [[ "$pos" -ne 10 ]] or test "$pos" -ne 10

while (( 10 - $pos ))
do

        # loop body ... consisting of
        # case statement processing inputs
        # followed by logic handling overshoot, snakes and ladders!

done
printf "\n*****************\n    Finished!    \n*****************\n"

exit 0
```

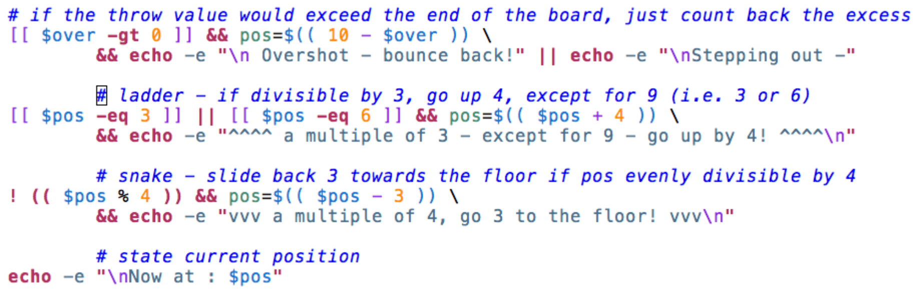
This is also an example of how one could 'comment in the algorithm', while developing the code,
which helps keep you on track while developing,
and leaves fully commented code when finished,
to aid debugging and subsequent maintenance or extension.

First part of loop body using case with break & continue

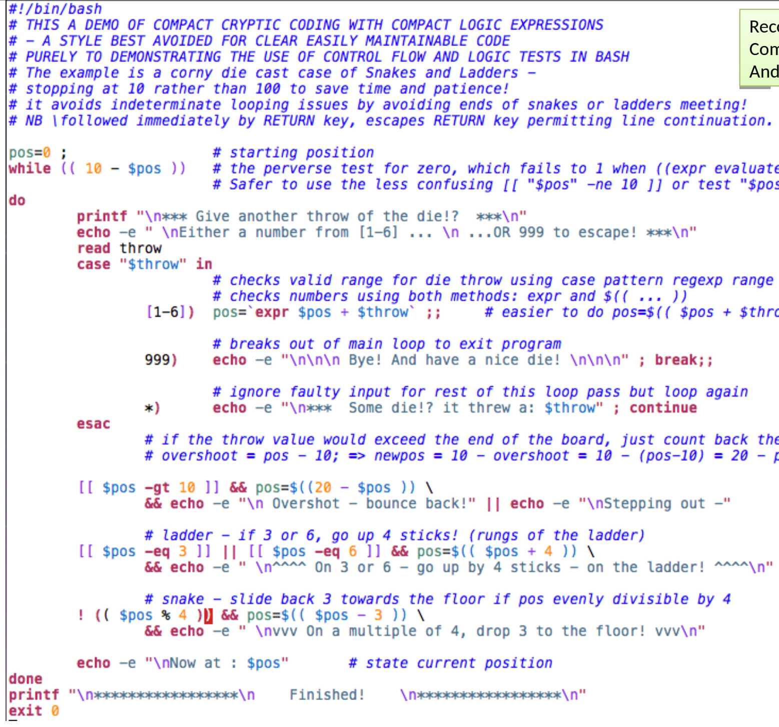


```
printf "\n*** Give another throw of the die!?  ***\n"
echo -e "\n*** Either a number from [1-6] ... \n ...OR 999 to escape! ***"
read throw
case "$throw" in

        # checks valid range for throw of die ...
        # ...using case pattern regexp range [1-6])
        # checks numbers using both methods: expr and $(( ... ))
        [1-6])  pos=`expr $pos + $throw`
                over=$(( $pos - 10 )) ;;

                # breaks out of main loop to finish
        999)    echo -e "\n\n\n Bye! And have a nice die! \n\n\n"
                break;;

                # ignore faulty input for rest of this pass
        *)      echo -e "\n***  Some die!? it threw a: $throw"
                continue
esac
```

Second part of loop body – logic for overshoot, snakes & ladders



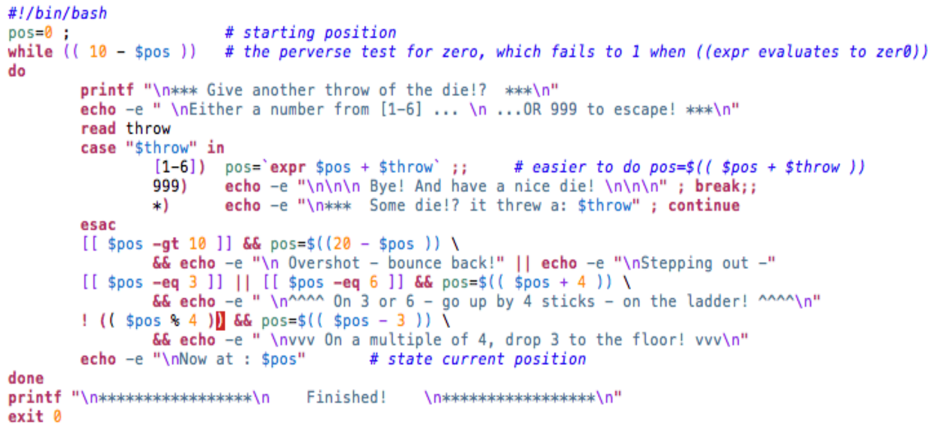
```
# if the throw value would exceed the end of the board, just count back the excess
[[ $over -gt 0 ]] && pos=$(( 10 - $over )) \
        && echo -e "\n Overshot - bounce back!" || echo -e "\nStepping out -"

        # ladder - if divisible by 3, go up 4, except for 9 (i.e. 3 or 6)
[[ $pos -eq 3 ]] || [[ $pos -eq 6 ]] && pos=$(( $pos + 4 )) \
        && echo -e "^^^^ a multiple of 3 - except for 9 - go up by 4! ^^^^\n"

        # snake - slide back 3 towards the floor if pos evenly divisible by 4
! (( $pos % 4 )) && pos=$(( $pos - 3 )) \
        && echo -e "vvv a multiple of 4, go 3 to the floor! vvv\n"

        # state current position
echo -e "\nNow at : $pos"
```



```
#!/bin/bash
# THIS A DEMO OF COMPACT CRYPTIC CODING WITH COMPACT LOGIC EXPRESSIONS
# - A STYLE BEST AVOIDED FOR CLEAR EASILY MAINTAINABLE CODE
# PURELY TO DEMONSTRATING THE USE OF CONTROL FLOW AND LOGIC TESTS IN BASH
# The example is a corny die cast case of Snakes and Ladders -
# stopping at 10 rather than 100 to save time and patience!
# it avoids indeterminate looping issues by avoiding ends of snakes or ladders meeting!
# NB \followed immediately by RETURN key, escapes RETURN key permitting line continuation.

pos=0 ;                 # starting position
while (( 10 - $pos ))   # the perverse test for zero, which fails to 1 when ((expr evaluates to zer0))
                        # Safer to use the less confusing [[ "$pos" -ne 10 ]] or test "$pos" -ne 10
do
        printf "\n*** Give another throw of the die!?  ***\n"
        echo -e " \nEither a number from [1-6] ... \n ...OR 999 to escape! ***\n"
        read throw
        case "$throw" in
                        # checks valid range for die throw using case pattern regexp range [1-6])
                        # checks numbers using both methods: expr and $(( ... ))
                [1-6])  pos=`expr $pos + $throw` ;;     # easier to do pos=$(( $pos + $throw ))

                        # breaks out of main loop to exit program
                999)    echo -e "\n\n\n Bye! And have a nice die! \n\n\n" ; break;;

                        # ignore faulty input for rest of this loop pass but loop again
                *)      echo -e "\n***  Some die!? it threw a: $throw" ; continue
        esac
                # if the throw value would exceed the end of the board, just count back the excess
                # overshoot = pos - 10; => newpos = 10 - overshoot = 10 - (pos-10) = 20 - pos

        [[ $pos -gt 10 ]] && pos=$((20 - $pos )) \
                && echo -e "\n Overshot - bounce back!" || echo -e "\nStepping out -"

                # ladder - if 3 or 6, go up 4 sticks! (rungs of the ladder)
        [[ $pos -eq 3 ]] || [[ $pos -eq 6 ]] && pos=$(( $pos + 4 )) \
                && echo -e " \n^^^^ On 3 or 6 - go up by 4 sticks - on the ladder! ^^^^\n"

                # snake - slide back 3 towards the floor if pos evenly divisible by 4
        ! (( $pos % 4 )) && pos=$(( $pos - 3 )) \
                && echo -e " \nvvv On a multiple of 4, drop 3 to the floor! vvv\n"

        echo -e "\nNow at : $pos"       # state current position
done
printf "\n******************\n    Finished!    \n******************\n"
exit 0
```

Recommended style;
Commented for clarity
And maintenance ease

Compact cryptic to fit on a slide for show!? ;-)



```
#!/bin/bash
pos=0 ;                 # starting position
while (( 10 - $pos ))   # the perverse test for zero, which fails to 1 when ((expr evaluates to zer0))
do
        printf "\n*** Give another throw of the die!?  ***\n"
        echo -e " \nEither a number from [1-6] ... \n ...OR 999 to escape! ***\n"
        read throw
        case "$throw" in
                [1-6])  pos=`expr $pos + $throw` ;;     # easier to do pos=$(( $pos + $throw ))
                999)    echo -e "\n\n\n Bye! And have a nice die! \n\n\n" ; break;;
                *)      echo -e "\n***  Some die!? it threw a: $throw" ; continue
        esac
        [[ $pos -gt 10 ]] && pos=$((20 - $pos )) \
                && echo -e "\n Overshot - bounce back!" || echo -e "\nStepping out -"
        [[ $pos -eq 3 ]] || [[ $pos -eq 6 ]] && pos=$(( $pos + 4 )) \
                && echo -e " \n^^^^ On 3 or 6 - go up by 4 sticks - on the ladder! ^^^^\n"
        ! (( $pos % 4 )) && pos=$(( $pos - 3 )) \
                && echo -e " \nvvv On a multiple of 4, drop 3 to the floor! vvv\n"
        echo -e "\nNow at : $pos"       # state current position
done
printf "\n******************\n    Finished!    \n******************\n"
exit 0
```

Merely for illustration of :-
- Control structures : while, case, break, continue, cryptic if then && etc.
- Logical test [[…]]
- Numerical evaluation and test ((….))

This programming style is too condensed, but is compact for presentation.

Exercise…

1. Replace both the shift values and associated logic in the main loop ..
   …Tricky but easy enough, if done right!
   a) Shift Values
      i. Ladder from 4 to 3
      ii. Snake /slide from 3 to 4
   b) Associated logic : so that repeated shifts may now follow;
      e.g. land on an 8, slide by 4 to 4, and then by another 4 to ground
      land on a 3, slide another 3 to 6 & another to 9
      (assume you can now slide +3 from a 9 with bounceback!)

      PS Do this program on your own, and if you notice any odd behaviour in the code, then keep it to yourself, as it may be part of the assignment…
      … of course, not all odd behaviour is attributable to the assignment!