The C-style for-loop

Synopsis

```
for (( <EXPR1> ; <EXPR2> ; <EXPR3> )); do
  <LIST>
done
```

```
# as a special case: without semicolon after ((...))
for (( <EXPR1> ; <EXPR2> ; <EXPR3> )) do
  <LIST>
done
```

```
# alternative, historical and undocumented syntax
for (( <EXPR1> ; <EXPR2> ; <EXPR3> )) {
  <LIST>
}
```

Description

The C-style for-loop is a compound command derived from the equivalent ksh88 feature, which is in turn derived from the C "for" keyword. Its purpose is to provide a convenient way to evaluate arithmetic expressions in a loop, plus initialize any required arithmetic variables. It is one of the main "loop with a counter" mechanisms available in the language.

The `((;;))` syntax at the top of the loop is not an ordinary arithmetic compound command, but is part of the C-style for-loop's own syntax. The three sections separated by semicolons are arithmetic expression contexts. Each time one of the sections is to be evaluated, the section is first processed for: brace, parameter, command, arithmetic, and process substitution/expansion as usual for arithmetic contexts. When the loop is entered for the first time, `<EXPR1>` is evaluated, then `<EXPR2>` is evaluated and checked. If `<EXPR2>` is true, then the loop body is executed. After the first and all subsequent iterations, `<EXPR1>` is skipped, `<EXPR3>` is evaluated, then `<EXPR2>` is evaluated and checked again. This process continues until `<EXPR2>` is false.

- `<EXPR1>` is to **initialize variables** before the first run.
- `<EXPR2>` is to **check** for a termination condition. This is always the last section to evaluate prior to leaving the loop.
- `<EXPR3>` is to **change** conditions after every iteration. For example, incrementing a counter.

If one of these arithmetic expressions in the for-loop is empty, it behaves as if it would be 1 (**TRUE** in arithmetic context).

Like all loops (Both types of `for` -loop, `while` and `until`), this loop can be:

- Terminated (broken) by the break builtin, optionally as `break N` to break out of `N` levels of nested loops.
- Forced immediately to the next iteration using the continue builtin, optionally as the `continue N` analog to `break N`.

The equivalent construct using a while loop and the arithmetic expression compound command would be structured as:

```
(( <EXPR1> ))
while (( <EXPR2> )); do
  <LIST>
  (( <EXPR3> ))
done
```

The equivalent `while` construct isn't exactly the same, because both, the `for` and the `while` loop behave differently in case you use the continue command.

Alternate syntax

Bash, Ksh93, Mksh, and Zsh also provide an alternate syntax for the `for` loop - enclosing the loop body in `{...}` instead of `do ... done`:

```
for ((x=1; x<=3; x++))
{
  echo $x
}
```

This syntax is **not documented** and shouldn't be used. I found the parser definitions for it in 1.x code, and in modern 4.x code. My guess is that it's there for compatibility reasons. Unlike the other aforementioned shells, Bash does not support the analogous syntax for case..esac.

Return status

The return status is that of the last command executed from `<LIST>`, or `FALSE` if any of the arithmetic expressions failed.

Alternatives and best practice

TODO: Show some alternate usages involving functions and local variables for initialization.

Examples

Simple counter

A simple counter, the loop iterates 101 times ("0" to "100" are 101 numbers → 101 runs!), and everytime the variable `x` is set to the current value.

- It **initializes** `x = 0`
- Before every iteration it **checks** if `x ≤ 100`
- After every iteration it **changes** `x++`

```
for ((x = 0 ; x <= 100 ; x++)); do
  echo "Counter: $x"
done
```

Stepping counter

This is the very same counter (compare it to the simple counter example above), but the **change** that is made is a `x += 10` . That means, it will count from 0 to 100, but with a **step of 10**.

```
for ((x = 0 ; x <= 100 ; x += 10)); do
  echo "Counter: $x"
done
```

Bits analyzer

This example loops through the bit-values of a Byte, beginning from 128, ending at 1. If that bit is set in the `testbyte` , it prints " `1` ", else " `0` " ⇒ it prints the binary representation of the `testbyte` value (8 bits).

```
#!/usr/bin/env bash
# Example written for http://wiki.bash-hackers.org/syntax/ccmd/c_for#
bits_analyzer
# Based on TheBonsai's original.

function toBin {
        typeset m=$1 n=2 x='x[(n*=2)>m]'
        for ((x = x; n /= 2;)); do
                printf %d $(( m & n && 1))
        done
}

function main {
        [[ $1 == +([0-9]) ]] || return
        typeset result
        if (( $(ksh -c 'printf %..2d $1' _ "$1") == ( result = $(toBi
n "$1") ) )); then
                printf '%s is %s in base 2!\n' "$1" "$result"
        else
                echo 'Oops, something went wrong with our calculatio
n.' >&2
                exit 1
        fi
}

main "${1:-123}"

# vim: set fenc=utf-8 ff=unix ft=sh :
```

Why that one begins at 128 (highest value, on the left) and not 1 (lowest value, on the right)? It's easier to print from left to right…

We arrive at 128 for `n` through the recursive arithmetic expression stored in `x`, which calculates the next-greatest power of 2 after `m`. To show that it works, we use ksh93 to double-check the answer, because it has a built-in feature for `printf` to print a representation of any number in an arbitrary base (up to 64). Very few languages have that ability built-in, even things like Python.

Up, down, up, down...

This counts up and down from `0` to `${1:-5}`, `${2:-4}` times, demonstrating more complicated arithmetic expressions with multiple variables.

```
for (( incr = 1, n=0, times = ${2:-4}, step = ${1:-5}; (n += incr) %
step || (incr *= -1, --times);)); do
    printf '%*s\n' "$((n+1))" "$n"
done
```

```
 ~ $ bash <(xclip -o)
 1
  2
   3
    4
     5
    4
   3
  2
 1
0
 1
  2
   3
    4
     5
    4
   3
  2
 1
```

Portability considerations

- C-style for loops aren't POSIX. They are available in Bash, ksh93, and zsh. All 3 have essentially the same syntax and behavior.
- C-style for loops aren't available in mksh.

Bugs

- *Fixed in 4.3*. ~~There appears to be a bug as of Bash 4.2p10 in which command lists can't be distinguished from the for loop's arithmetic argument delimiter (both semicolons), so command substitutions within the C-style for loop expression can't contain more than one command.~~

See also

- Internal: Arithmetic expressions
- Internal: The classic for-loop
- Internal: The while-loop

Discussion

Martin Kealey, 2013/06/06 02:43 ()

It should be noted that the equivalence with the while-loop is approximate; it differ, in that "continue" in a for-loop will cause the 3rd ("increment") expression to be evaluated.