Pediluve — Tutorial D2 PM

version **#dirty**

Assistants C/Unix 2024 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

*https://intra.forge.epita.fr

1 Loops

You have seen how to test conditions but you had no way of repeating commands while a certain condition is true or false. This is what `while`, `until` and `for` loops allow you to do.

1.1 While loops

1.1.1 The basics

`while` loops let you execute commands *while* a certain condition is true. `until` loops let you do the same, but *until* a certain condition becomes true.

```
while command; do
    do_something
done

until command; do
    do_something
done
```

Here you can see a simple `while` example:

```
42sh$ n=0
42sh$ while [ $n -le 5 ]; do echo $((n++)); done
0
1
2
3
4
5
42sh$
```

Tips

In order to parse a file line by line, you need to use the `IFS`:

```
while IFS='' read -r line; do
    # do something with $line
done < "file.txt"
```

Note the last line, you will see the purpose of the < later, just do not forget it!

1.2 For loops

1.2.1 The basics

`for` loops let you iterate through a list with a variable that will successively take every value of the list.

```
for var in list; do
    do_something
done
```

A more concrete example:

```
42sh$ cat loop.sh
#!/bin/sh

for i in item1 item2 item3; do
    echo "$i"
done
```

```
42sh$ ./loop.sh
item1
item2
item3
```

This is a very simple `for` loop, but you can much more with them.

1.2.2 `seq` command

You can use *command substitution* to generate a list to iterate on. For instance, the `seq` binary generates a list of integers, and can be used to do `for` loops.

```
#!/bin/sh

for i in $(seq 2 5); do
    echo "number: $i"
done
```

```
42sh$ ./seq_loop.sh
number: 2
number: 3
number: 4
number: 5
```

Tips

You can also omit the first boundary so that it starts from one.

```
42sh$ seq 3
1
2
3
```

You can even specify the start, step, and end. Much like the `range` function in Python.

```
42sh$ seq 0 10 42
0
10
20
30
40
```

1.2.3 Omitting the list

While inside a shell script, omitting the list will cause the loop to run on '`$@`'.

```sh
#!/bin/sh

for arg; do
    echo "$arg"
done
```

```
42sh$ ./no_list.sh arg1 2 hello arg4
arg1
2
hello
arg4
```

1.2.4 Exercise

Write a script where you have to iterate on each argument and print them if they are executable.

```
42sh$ ls
file1.txt file2.txt script.sh
42sh$ ./script.sh file1.txt /bin/sh file2.txt script.sh
/bin/sh
script.sh
```

1.3 Loop control

There are times when you want to control what happens in your loops. The shell provides you with two builtins: `break` and `continue`. Both of these builtins can take a parameter (set to 1 if not specified) and will control how the loop behaves. They are also very handy when you are dealing with nested loops.

1.3.1 `break`

The builtin `break` will terminate (jump out of) the current loop. The optional parameter indicates how many levels of loop the `break` should terminate.

```
#!/bin/sh

for i in $(seq 5); do
    if [ "$i" -eq 3 ]; then
        break
    fi
    echo "$i"
done
```

```
42sh$ ./loop_break.sh
1
2
```

```
#!/bin/sh

for i in $(seq 5); do
    for j in $(seq 5); do
        for k in $(seq 5); do
            if [ "$k" -gt 2 ]; then
                break 2
            fi
            echo "$i,$j,$k"
        done
    done
done
```

```
42sh$ ./loop_break.sh
1,1,1
1,1,2
2,1,1
2,1,2
3,1,1
3,1,2
4,1,1
4,1,2
5,1,1
5,1,2
```

1.3.2 `continue`

The builtin `continue` will cut the current iteration and begin the next one. The optional parameter indicates which loop the next iteration should be done from. The construct `continue n` is to be avoided as it generally makes the code really difficult to understand.

```
#!/bin/sh

for i in $(seq 10); do
    if [ "$i" -lt 5 ]; then
        continue
    fi
    echo "$i"
done
```

```
42sh$ ./loop_continue.sh
5
6
7
8
9
10
```

Tips

If a loop seems to run infinitely just hit `Ctrl + C` keys, it will stop the running program.

1.4 Exercise: My Sleep

1.4.1 Goal

The binary `sleep` will delay for the number of seconds specified in parameter.

Write a script that takes one parameter and sleeps (without using `sleep`) for the correct amount of time thanks to a `while` loop and a `break`.

2 Arithmetic

2.1 Syntax

Arithmetic calculations can be done with the special syntax `((calculation))`.

```
42sh$ ((a = 3, b = 2, a += b)) # shell arithmetic has a flexible syntax
42sh$ echo "$a"
5
```

Add a dollar to this syntax and it is called *arithmetic expansion*:

```
42sh$ echo "$a" + 1 # won't work
5 + 1
42sh$ echo "$((a + 1))" # but with arithmetic expansion ...
6
42sh$ echo "$a" # 'a' variable value is unchanged
5
```

Something important to note is that arithmetic operations have some special behaviors:

- Unset and null variables, or any variable containing something other than a numerical value are considered to be equal to zero when used in an arithmetical context.

  ```
  42sh$ a='hello'
  42sh$ echo "$((a + 1))"
  1
  42sh$ echo "$a"
  hello
  ```

- The dollar notation is not needed for variables inside arithmetic expansion and is not recommended unless necessary as empty variables will not be replaced by 0 after variable expansion.

  ```
  42sh$ a=''
  42sh$ echo "$((a * 2))" # no dollar, 'a' is replaced by 0
  0
  42sh$ echo "$(($a * 2))" # with a dollar, '$a' is expanded to... nothing
  bash: * 2: syntax error: operand expected (error token is "* 2"))
  ```

  By adding a dollar, the variable is expanded before arithmetic expansion, hence when arithmetic expansion takes place, the line looks like this: `$(( * 2))`. However, the `*` needs two operands as specified in the error output, leading to an error.

- You will see later that arithmetic expansion can be used for test operations. Indeed `((0))` has a 1 return code (`false`), and anything different from 0 has a 0 return code (`true`).

2.2 Operator precedence

Here is a list of all available operators for arithmetical operations, sorted from the highest to the lowest priority.

Name	Meaning
`var++ var--`	post-increment, post-decrement
`++var --var`	pre-increment, pre-decrement
`! ~`	negation logical, bitwise negation
`**`	exponentiation
`* / %`	multiplication, division, modulo
`+ -`	addition, subtraction
`<< >>`	left, right shift bitwise
`< > <= >=`	comparison
`== !=`	equality, inequality
`&`	bitwise AND
`^`	bitwise XOR
`|`	bitwise OR
`&&`	logical AND
`||`	logical OR
`?:`	ternary operator
`=`	assignment
`*= /= %= += -= <<= >>= &=`	times-equal, divide-equal, mod-equal, etc.
`,`	links a sequence of operations

Be careful!

You will rarely encounter the syntax `$[calculation]`, which is a deprecated notation for arithmetic expansion. Do **not** use it.

Going further...

- The first notation of arithmetic without the dollar is not specified by POSIX and will work on `bash`/`ksh` but will produce an error on `dash` for instance.
- Float operations are not supported by `bash`, for advanced arithmetical calculations, use the command `bc`. However, `ksh` natively supports float calculation with arithmetic expansion.
- You might encounter `expr` (needed for pre-POSIX Bourne-like shells) or `let` notations which are both similar to `$(())` but are still less portable. On the other hand, `csh`/`tcsh` use a builtin arithmetic operator: `@ a = 2 * 3`.

  For example, here are some uses of `expr`:

```
42sh$ echo $((21 * 2))
42
42sh$ expr 21 \* 2
42
42sh expr 42 / 2
21
```

Be careful!

Do not forget to escape the `*` operator. Otherwise, it will result in an error since the `*` is a shell glob.

2.3 Exercise: Format Seconds

2.3.1 Goal

Write a script that prints a formatted output of a number of seconds passed as argument. No need to convert minutes to hours.

```
42sh$ ./format_seconds.sh 80
1:20
42sh$ ./format_seconds.sh 185
3:05
42sh$ ./format_seconds.sh 12345
205:45
```

3 `PATH`

3.1 Concept

You used a lot of different commands. But do you know how your shell manages to find these programs?

The `PATH` is an environment variable (`man 1 env`), that tells your shell where are stored the *executables*.

You can display this environment variable by typing:

```
42sh$ echo $PATH
```

As you can see, the `PATH` contains the different locations where your shell is going to look for executables separated by a colon (`:`).

It means that when you type:

```
42sh$ ls
```

Your shell is going to search in all the directories contained in your `PATH` for an executable named `ls`.

Tips

The path you will see in your terminal is not necessarily the same as the one in our example. For the rest of the commands, you will need to adapt it to your path.

If you want to see where a program is located, you can just do:

```
42sh$ whereis ls
ls: /nix/store/l11am15d91wpragk6c63qmkq9j98n5vq-system-path/bin/ls
```

Note that you can call a program directly with its absolute path, like this:

```
42sh$ /nix/store/l11am15d91wpragk6c63qmkq9j98n5vq-system-path/bin/ls
```

To launch an executable located in the current directory instead of the `PATH`, you have to use `./` before the name of the file. This is called the relative path.

For instance:

```
42sh$ cp /nix/store/l11am15d91wpragk6c63qmkq9j98n5vq-system-path/bin/ls .
42sh$ ./ls
```

This calls `ls` in the current directory (not the one in the `PATH`).

Tips

Absolute path designates a unique location in the filesystem regardless of the current working directory. It always begins with a '/'.

Relative path designates a path relative to your current directory.

Remember that `.` is your current directory, and `..` refers to the parent directory.

3.2 Exercise

1. Create the directory `bin` in your current directory.
2. Copy the `cp` program into `bin`.
3. Set the `PATH` variable to `bin`. You can achieve this by typing:

```
42sh$ PATH=/path/to/your/directory/bin
```

Be careful!

Use the absolute path to the local bin directory otherwise if you change your current directory it will not work anymore.

4. Try to use `ls`. What is happening?
5. Copy the `ls` program to the `bin` directory.
6. Try to use `ls`. What is happening?

Going further...

On other Linux distributions, you can source your shell configuration file to restore your `PATH`, with the command `source /etc/profile`. However, using the school's PIE, it is easier to open a new terminal.

4 Command substitution

4.1 Subshell

A subshell, as its name suggests, allows you to execute code in another context, more precisely in another shell process. The syntax is pretty simple, you just need to nest your code between a pair of parentheses.

```
42sh$ var='Current shell'
42sh$ (var='Subshell'; echo "$var") # assign a value to var and print it
Subshell
42sh$ echo "$var"
Current shell
```

As you can see, the value of `var` was not modified after leaving the subshell. This introduces a kind of scoping for variables even though all variables are global by default.

Another thing to note is that the current environment is passed to subshells so you do not need to export any variables. On the other hand, all environment changes happening in a subshell have absolutely zero effect on the shell that launched it.

But if it has no effect, what is the point?

4.2 Command substitution

Command substitution is a way to reassign the output of a subshell to something else. Two syntaxes are available:

- Just add a dollar before your subshell parenthesis: `$(command)`
- Use backquotes: `` `command` ``

Be careful!

Backquotes are an older notation, the dollar notation is now preferred as it allows nesting (backquotes need to be escaped in order to be nested). Always use the dollar notation.

```
42sh$ cat cool_commands.txt
ls
cp
man
42sh$ first_command_name="$(head -n 1 cool_commands.txt)"
42sh$ echo "$first_command_name"
ls
42sh$ "$(tail -n 1 cool_commands.txt)" --help # run help for the last item
Usage: man [OPTION...] [SECTION] PAGE...
...
42sh$ echo "$(echo A"$(echo C"$(echo U)")")" # nested
ACU
```

Going further...

You may have noticed the heavy quoting on the last command. `IFS` field splitting also applies to command substitution, so with a default `IFS` you will not have any issues without any quoting. However, with an `IFS='ACU'`, nothing would be printed.

I must not fear. Fear is the mind-killer.