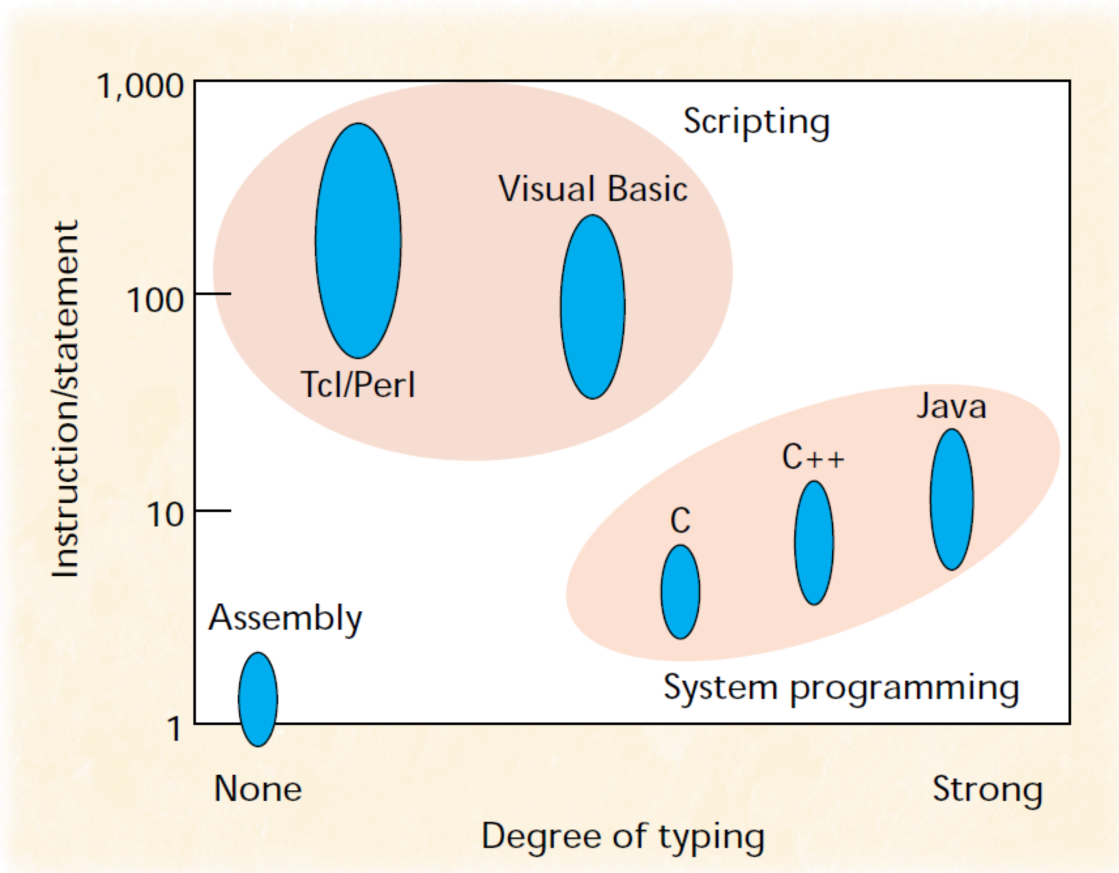
Bash Tutorial

Eigil Obrestad and Erik Hjelmås

August 18, 2015



(OUSTERHOUT, J., "Scripting: Higher-Level Programming for the 21st Century", IEEE Computer, Vol. 31, No. 3, March 1998, pp. 23-30.)

From Ousterhout, 1998:
While programming languages like C/C++ are designed for low-level construction of data structures and algorithms, scripting languages are designed for high-level "gluing" of existing components. Components are created with low-level languages and glued together with scripting languages.

WARNING!

The following presentation is NOT meant to be a comprehensive/complete tour of the Bash language.

The purpose is to get you started with some basic program constructions which you will recognize based on some-sort-of-programming-background.

At the end of the presentation you will find pointers to more comprehensive material.

Practice

You need a GNU/Linux distribution (e.g. Ubuntu) running on a physical or virtual machine with working access to the internet, and with `wget` installed.

Log in and open a terminal window, download the examples as we go along with

```
wget http://www.ansatt.hig.no/erikh/tutorial-bash/FILENAME
```

(or download all at once with filename `bash-examples.tar`)

You will find the `FILENAME` on the second line of each example. For each example do

1. Download
   `wget http://www.ansatt.hig.no/erikh/tutorial-bash/FILENAME`
2. View the code
   `cat FILENAME` or `less FILENAME`
3. Execute the code
   `bash FILENAME`
   or (make it executable with `chmod +x FILENAME`)
   `./FILENAME`

It is easy to write bash scripts, but sometimes your scripts will behave strangely. This is due to the fact that there are many pitfalls in Bash. It is very easy to write statements that appear logical to you in your way of thinking programming, but due to the nature of a shell environment such as Bash, they will not produce the expected results. I strongly recommend that you quickly browse (and remember as a reference) the following excellent document:
`http://mywiki.wooledge.org/BashPitfalls`
and use the following tool to check the quality of your script (you can also probably apt-get install shellcheck):
`http://www.shellcheck.net/`

Hello World

```
#!/bin/bash
# hello.bash

echo "Hello world!"
```

make executable and execute:

```
chmod +x hello.bash
./hello.bash
```

The SheBang/HashBang `#!` is treated as a comment by the interpreter, but it has a special meaning to the operating system's program loader (the code that is run when one of the `exec` system calls are executed) on Unix/Linux systems. The program loader will make sure this script is interpreted by the program listed after the `#!`

Since Unix/Linux does not use file endings for identifying the file, there is no reason for us to do so in scripts either. The OS knows from the SheBang/HashBang what interpreter to use. However during the development of a script it can be nice to know something about the file content based on the file ending, so it is common to use `.bash` or `.sh` endings. Since this tutorial is about the Bash language, including Bash-specific features that are not POSIX-compliant, we stick with `.bash` as file endings.

1 Variables

Single Variables

```
#!/bin/bash
# single-var.bash

firstname=Mysil
lastname=Bergsprekken
fullname="$firstname $lastname"
echo "Hello $fullname, may I call you $firstname?"
```

A single variable is not typed, it can be a number or a string.

Do not put spaces before or after = when assigning values to variables.

If you need to substitute a variable immediately before a string, e.g the variable `sum` and the string `KB`, use curly braces around the variable (you will have to do this every time for array elements as well as you will see below):
`echo "disk usage is ${sum}KB"`

Scope: variables are global unless specified inside a block and starting with the keyword `local`.

(in general, use lower case variable names, upper case implies it's a SHELL/ENVIRONMENT variable)

Single and Double Quotes

```
#!/bin/bash
# quotes.bash

name=Mysil
echo Hello   $name
echo "Hello   $name"
echo 'Hello   $name'
```

Variables are expanded/interpolated inside double quotes, but not inside single quotes. We use double quotes when we have a string.

1.1 Arrays

Arrays

Bash supports simple one-dimensional arrays

```
#!/bin/bash
# array.bash

os=('linux' 'windows')
os[2]='mac'
echo "${os[1]}"  # print windows
echo "${os[@]}"  # print array values
echo "${!os[@]}" # print array indices
echo "${#os[@]}" # length of array
```

Automatic expansion of arrays (automatic declaration and garbage collection). `os[2]='mac'` can also be written as `os+=('mac')`

Associative Arrays

```
#!/bin/bash
# assoc-array.bash

declare -A user         # must be declared
user=(                        \
       [frodeh]="Frode Haug" \
       [ivarm]="Ivar Moe"    \
     )
user[lailas]="Laila Skiaker"
echo "${user[ivarm]}"  # print Ivar Moe
echo "${user[@]}"      # print array values
echo "${!user[@]}"     # print array indices (keys)
echo "${#user[@]}"     # length of array
```

Associative arrays were introduced with Bash version 4 in 2009. If we don't declare the variable as an associative array with `declare -A` before we use it, it will be an ordinary indexed array.

`user[lailas]="Laila Skiaker"` can also be written as
`user+=([lailas]="Laila Skiaker")`

1.2 Structures/Classes

Structures/Classes

Sorry, no structs or classes in Bash ...

1.3 Command-line args

Command-Line Arguments

Scriptname in $0, arguments in $1, $2, ...

```
#!/bin/bash
# cli-args.bash

echo "I am $0, and have $# arguments \
      first is $1"
```

Bash accepts the first nine arguments as `$1`...`$9`, for further arguments use `${10}`, `${11}`, ...

2 Input

2.1 Input

Input From User

```
#!/bin/bash
# input-user.bash

echo -n "Say something here:"
read -r something
echo "you said $something"
```

Input From STDIN

Same way, commonly without an echo first

```
#!/bin/bash
# input-stdin.bash

read -r something
echo "you said $something"
```

can be executed as

```
echo "hey hey!" | ./input-stdin.bash
```

Of course, input from user is from `STDIN`.

We use `read -r` to avoid `read` removing backslashes in the input.

2.2 System commands

Input from System Commands

You can use $(cmd) (supports nesting) or `cmd` (deprecated)

```
#!/bin/bash
# input-commands.bash

kernel="$(uname -sr)"
echo "I am running on $kernel in $(pwd)"
```

This is also called *command substitution*. `...` (backticks) is depreciated because it's difficult to read, and can create some problems, see http://mywiki.wooledge.org/BashFAQ/082

3 Conditions

3.1 if/else

if/else

```
#!/bin/bash
# if.bash

if [[ "$#" -ne 1 ]]; then
   echo "usage: $0 <argument>"
fi
```

Note: there must be spaces around [[and]].

There is also an older (slower) and more portable (meaning POSIX defined) operator, [which is actually an alias for the operator test, meaning

```
[ "$#" -ne 2 ]
# is the same as
test "$#" -ne 2
```

3.2 Operators

Arithmetic Comparison

Operator	Meaning
-lt	Less than
-gt	Greater than
-le	Less than or equal to
-ge	Greater than or equal to
-eq	Equal to
-ne	Not equal to

String Comparison

Operator	Meaning
<	Less than, in ASCII alphabetical order
>	Greater than, in ASCII alphabetical order
=	Equal to
==	Equal to
!=	Not equal to

File Tests

Operator	Meaning
−e	Exists
−s	Not zero size
−f	Regular file
−d	Directory
−l	Symbolic link
−u	Set-user-id (SetUID) flag set

There are many more file test operators of course.

Boolean

Operator	Meaning
!	Not
&&	And
||	Or

Numerical or String Compare

```
#!/bin/bash
# if-num-string.bash

if [[ "$#" -ne 2 ]]; then
   echo "usage: $0 <argument> <argument>"
   exit 0
elif [[ "$1" -eq "$2" ]]; then
   echo "$1 is arithmetic equal to $2"
else
   echo "$1 and $2 arithmetic differs"
fi
if [[ "$1" == "$2" ]]; then
   echo "$1 is string equal to $2"
else
   echo "$1 and $2 string differs"
fi
if [[ -f "$1" ]]; then
   echo "$1 is also a file!"
fi
```

This shows the if-elif-else construction, the difference between string and numerical comparison, and a file test operator.

Note the difference between −eq and ==

```
$ ./if-num-string.bash 1 01
1 is arithmetic equal to 01
1 and 01 string differs
```

Boolean example

```
#!/bin/bash
# if-bool.bash

if [[ 1 -eq 2 && 1 -eq 1 || 1 -eq 1 ]]; then
   echo "And has precedence"
else
   echo "Or has precedence"
fi

# force OR precedence:

if [[ 1 -eq 2 && (1 -eq 1 || 1 -eq 1) ]]; then
   echo "And has precedence"
else
   echo "Or has precedence"
fi
```

AND is always (as known from mathematics courses) evaluated before OR (binds more tightly). Write it down in logic (truth table) if you are unsure.

3.3 Switch/case

Case

```
#!/bin/bash
# switch.bash

read -r ans
case $ans in
yes)
   echo "yes!"
   ;;&             # keep testing
no)
   echo "no?"
   ;;              # do not keep testing
*)
   echo "$ans???"
   ;;
esac
```

See also select and whiptail.

4 Iteration

4.1 For

For loop

```
#!/bin/bash
# for.bash

for i in {1..10}; do
   echo -n "$i "
done
echo

# something more useful:

for i in ~/*; do
   if [[ -f $i ]]; then
      echo "$i is a regular file"
   else
      echo "$i is not a regular file"
   fi
done
```

We can also use `for i in $(ls -1 ~/)` or `for i in $(stat -c "%n" ~/*)` but this creates problems if we have filenames with spaces, so it's much better to use Bash' builtin expansion operator `*` as we do in this example. See more on this below when wew talk about `while`.

If you just quickly want to iterate over a short list of fixed numbers, `{1..10}` is ok, but this has two downsides: 1. these numbers are generated at once, so this consumes memory if the list is long, 2. if you want to iterate up to a variable `$max` this does not work and you shoud insted use the more traditional programming syntax
`for ((i=1; i<$max; i++))`

4.2 While

In general, we prefer using a while loop instead of a for loop when iterating over a set of items (items being files, process names, user names, lines in a file, etc). This is due to the fact that when using a for loop as shown above, the list of items is generated beforehand and thereby can consume significant amounts of memory. A while loop allows us to iterative item by item without pre-generating all the items in memory.

While

We want to read from STDIN and do stuff line by line

```
#!/bin/bash
# while.bash
```

```
i=0
while read -r line; do
   foo[i]=$line
   ((i++))
done
echo "i is $i, size of foo ${#foo[@]}"
```

```
$ ls -1 | ./while.bash
i is 20, size of foo is 20
```

A problem ...

What if we want to pipe into a while inside our script:

```
#!/bin/bash
# while-pipe-err.bash

i=0
ls -1 | while read -r line; do
   foo[i]=$line
   ((i++))
done
echo "i is $i, size of foo ${#foo[@]}"
```

```
$ ./while-pipe-err.bash
i is 0, size of foo is 0
```

In other words, this does not work due to a subshell being used (because of the pipe) inside while!

Meaning that the variables outside the while loop are not accessible inside the while loop since it is run as a new process.

Proper Solution

Inspired by `http://mywiki.wooledge.org/BashGuide/Arrays`

```
#!/bin/bash
# while-pipe.bash

i=0
while read -r -d ''; do
   foo[i]=$REPLY
   ((i++))
done < <(find . -maxdepth 1 -print0)
echo "i is $i, size of foo ${#foo[@]}"
```

```
$ ./while-pipe.bash
i is 20, size of foo is 20
```

`while-pipe.bash` is the proper way of processing output from commands, note

- `-d` is delimiter which we set to the empty string (NUL-byte, `\0`)
- if we don't give a variable name to `read` it will place contents in the default variable `REPLY`
- `<(...)` is process substitution meaning it's the same as `$(...)` but we pipe in the output as well into the redirection `<` which "sends" the output into the loop.
- `find . -maxdepth 1 -print0` is the safe way of doing `ls`, `-maxdepth 1` limits find to only list files in `./` and `-print0` inserts the NUL-byte (`\0`) as a separator between the file names to avoid any confusion with odd characters in file names (the NUL-byte is not allowed in file names)

We can also solve this problem (if the problem is only processing files in the directory, note that the solution above with while is more general and can be used by any command capable of inserting a NUL-byte separator in its output) by rewriting it as a for loop instead (but, as mentioned, this has the downside of consuming memory if the list of files is big since it is generated before the loop initiates):

Solution with for instead of while

```
#!/bin/bash
# for-commands.bash

i=0
for line in ./*; do
   foo[i]=$line
   ((i++))
done
echo "i is $i, size of foo ${#foo[@]}"
```

5 Math

Operators

Operator	Meaning
+	Add
−	Subtract
*	Multiply
/	Divide
%	Modulus

Only on integers!

```
#!/bin/bash
# math.bash

i=0
((i++))
echo "i is $((i-2))" # print -1
```

A trick for floats

```
#!/bin/bash
# math-float.bash

echo "3.1+5.6 is $(echo '3.1+5.6' | bc)"
```

6 Functions

Functions

```
#!/bin/bash
# func.bash

# declare:
function addfloat {
   echo "$1+$2" | bc
}
# use:
addfloat 5.12 2.56
```

7 RegExp

Regular expressions intro 1/5

Special/Meta-characters:

`\ | ( ) [ ] { } ^ $ * + ? .`

These have to be protected with \, e.g. `http://www\.hig\.no`

> To match `c:\temp`, you need to use the regex `c:\\temp`. As a string in C++ source code, this regex becomes `"c:\\\\temp"`. Four backslashes to match a single one indeed.

(from `http://www.regular-expressions.info/characters.html`):

Regular expressions are the generic way to describe a string/syntax/word that you use for either syntax/compliance checking or searching. It is commonly used in configuration files. Think of it as a generic way of doing advanced search. Google would probably prefer user to only enter regular expression as search terms, but that would be to hard for the general population so Google offers "advanced search" instead:
`http://www.google.com/advanced_search`

There are many different regular expression engines, which differs mostly in features and speed. In this tutorial we will try to stick with simple examples which will the same in most engines (perl, pcre, extended posix, .NET, ...).

Regular expressions intro 2/5

Describing characters:

Operator	Meaning
`.`	Any single character
`[abcd]`	One of these characters
`[^abcd]`	Any one but these characters
`[A-Za-z0-9]`	A character in these ranges
`:word:`	A word (A-Za-z0-9_)
`:digit:`	A digit

`\w` is the same as `[a-zA-Z0-9]` and `\d` is the same as `[0-9]` . Many more of course ...

Regular expressions intro 3/5

Grouping:

Operator	Meaning
`()`	Group
`|`	OR

Anchoring:

Operator	Meaning
`^`	Beginning of line
`$`	End of line

Regular expressions intro 4/5

Repetition operators/Modifiers/Quantifiers:

Operator	Meaning
`?`	0 or 1 time
`*`	0 or more times
`+`	1 or more times
`{N}`	N times
`{N,}`	At least N times
`{N,M}`	At least N but not more than M

Demo: example with `cat a.html | egrep REGEXP` (four steps).

Regular expressions intro 5/5

Finding URLs in HTML: `(mailto|http):[^"]*`

Each line should be an email address: `^[A-Za-z0-9._-]+@[A-Za-z0-9.-]+$`

Remember that regexp engines are most often greedy, they try to match as much as possible, so using e.g. `.*` might match more than you were planning for.

7.1 Bash example

Bash example

```
#!/bin/bash
# regexp.bash

while read -r line; do
   if [[ $line =~ \
      ^[A-Za-z0-9._-]+@([A-Za-z0-9.-]+)$ ]]
   then
      echo "Valid email ${BASH_REMATCH[0]}"
      echo "Domain is ${BASH_REMATCH[1]}"
   else
      echo "Invalid email address!"
   fi
done
```

When we use regular expressions inside scripts, it is very useful to be able to extract parts of the match. We can do this by specifying the part with `(part)` and refer to it later in the `$BASH_REMATCH` array (if we specify two parts, the second one will be in `$BASH_REMATCH[2]` etc).

If you for some reason (maybe optimization) do not want the part inside parenthesis to be stored in `$BASH_REMATCH`, you can do this by saying `(?:part)`. In other words, the two characters `?:` here has the special meaning ”do not include what this parenthesis matches in `BASH_REMATCH`”.

Of course you can use regexp in many different components which you can include in your bash script (sed, grep, perl, ...).

8 Bash only

Advanced stuff

See Advanced Bash-Scripting Guide at

`http://tldp.org/LDP/abs/html/`

for everything you can do with Bash

9 Credits

Credits

J. Ousterhout, "Scripting: Higher-Level Programming for the 21st Century," IEEE Computer, Vol. 31, No. 3, March 1998, pp. 23-30. http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html http://www.linuxconfig.org/Bash_scripting_Tutorial http://www.thegeekstuff.com/2010/06/bash-array-tutorial/ http://www.panix.com/~elflord/unix/bash-tute.html http://www.codecoffee.com/tipsforlinux/articles2/043.html http://tldp.org/LDP/abs/html/ http://linuxsig.org/files/bash_scripting.html http://mywiki.wooledge.org/BashGuide http://www.regular-expressions.info/ Alexander Berntsen