

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXPAROLA 30
#define MAXRIGA 80
```

```
int main(int argc, char *argv[])
```

```
{
    int freq[MAXPAROLA]; /* vettore di contatori
    delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE *f;
```

```
for(i=0; i<MAXPAROLA; i++)
    freq[i]=0;
```

```
if(argc != 2)
```

```
{
    fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
}
```

```
f = fopen(argv[1], "r");
if(f==NULL)
```

```
{
    fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
}
```

```
while( fgets( riga, MAXRIGA, f ) != NULL )
```



UNIX/Linux Operating System

Shell scripts

Stefano Quer, Pietro Laface, and Stefano Scanzio

Dipartimento di Automatica e Informatica

Politecnico di Torino

skenz.it/os

stefano.scanzio@polito.it

Introduction to shell scripts

- ❖ Shell languages are **interpreted** languages
 - There is no explicit compilation
- ❖ Pros & Cons
 - Shell available in every UNIX / Linux environment
 - Faster production cycle
 - Lower run-time efficiency
 - Fewer debugging possibilities
- ❖ Used to write software
 - "Quick and dirty"
 - Sometimes a prototype, which is then translated into a low-level language such as C

Introduction to scripts

❖ BASH vs. Python (and other)

➤ Choice

- The main strength of BASH with respect to other languages (python, ruby, lua, etc.) is its ubiquity
- If the number of code lines is less than 100, it is better to choose BASH, otherwise Python

Introduction to scripts

❖ BASH vs. Python (and other)

➤ Performance

- To have high performance write a program not a script
- The BASH interpreter is very fast to start (starting phase)
- If you need to manipulate ASCII files, or heavily use shell commands or filters like sort, uniq, etc., BASH is more suitable and faster ("will smoke Python performance wise")
- If you need to manipulate floating point numbers Python is convenient ("will win hands down")

Introduction to shell scripts

❖ Scripts

- Are normally stored in files with **.sh** extension (or **.bash**)
 - But recall that the extensions are not used UNIX/Linux to determine the file type
- ❖ They can be executed using two techniques
 - Direct execution
 - Indirect execution

Direct execution

```
./scriptname args
```

- ❖ The script is executed from the command line as a normal executable file
 - The script file must have the **execute** permission
 - `chmod +x ./scriptname`
 - The first line of the script can specify the name of the script interpreter
 - `#!/bin/bash` or `#!/bin/sh`
 - It is possible to execute the script using a specific shell
 - `/bin/bash ./scriptname args`

Direct execution

```
./scriptname args
```

- ❖ The script is executed by a sub-shell
 - i.e., by a new shell process
 - Environment (variables) of the original process and of the new one are not the same
 - Changes to the environment variables made by the script, and used within the script, are lost at exit

Indirect execution

```
source ./scriptname args
```

- ❖ The source command executes the script given as its argument
 - It is the **current shell** to run the script
 - "The current shell sources the script"
 - It is not necessary that the script is executable
 - The changes made by the script to environment variables remain in effect in the current shell

Example: direct and indirect execution

Direct execution:

> scriptName.sh<return>

The shell executes the script as a sub-shell. Executing **exit** the sub-shell terminates. **The initial process resumes control.**

```
#!/bin/bash  
# NULL Script  
exit 0
```

indicates a comment

Indirect execution :

> source scriptName.sh<return>

The shell executes the script. Executing **exit** **the shell process terminates** (i.e., you kill the starting/original shell)

Script debugging

- ❖ There are not specific tools to debug bash scripts
 - It is obviously always possible to add explicit "echo"
- ❖ However, it is possible to "debug" a script in the following way
 - Full (the whole script)
 - It is obtained by indicating a "debug" option at the level of the entire script
 - Partial (only a few lines of the script)
 - It is obtained by indicating a "debug" option at the level of some lines of the script using the **set** command

Script debugging

❖ Possible options for both partial and full debug

➤ -o noexec, -n

- Executes a syntactic check, but the script is not executed

➤ -o verbose, -v

- Displays the executed commands

➤ -o xtrace, -x

- Displays the execution trace of the entire script

➤ -o nounset, -u

- Prints a error for undefined variables

Script debugging

❖ Fully debug

➤ From a shell command

- `/bin/bash -n ./scriptname args`

➤ Inside the script

- `#!/bin/bash -v`
- `#!/bin/bash -x`
- ...

❖ Partial debug

- `set -o verbose ... set +o verbose`
- `set -v ... set +v`
- `set -x ... set +x`

Syntax: general rules

- ❖ The bash language is relatively "high level", and it allows to mix
 - Standard shell commands
 - `ls`, `wc`, `find`, `grep`, ...
 - Standard constructs of the shell language
 - Input and output variables and parameters, operators (arithmetic, logic, etc.), control constructs (conditional, iterative), arrays, functions, etc.
- ❖ Often instructions/commands are written in separate lines
 - on the same line, they must be separated by `' ; '`

Syntax: general rules

❖ Comments

- Character **#** indicates the presence of a comment on the line
- A comment begins by character **#** and terminates at the end of line

❖ **exit** allows terminating a script returning an error code

- **exit**
- **exit [0|1|2|3|...]**
 - In shell, 0 means TRUE

Example of shell commands

Absolute path

```
#!/bin/bash
```

```
# This line is a comment
```

```
rm -rf ../../newDir/  
mkdir ../../newDir/  
cp * ../newDir/  
ls ../newDir/ ;
```

';' superfluous

```
# 0 is TRUE in shell programming
```

```
exit 0
```

From the calling shell:
echo \$?
returns 0

Arguments

- ❖ The arguments of the command line passed to the script are identified by \$
- ❖ Positional parameters
 - \$0 is the script name
 - \$1, \$2, \$3, ... indicate the arguments passed to the script on the command line
- ❖ Special parameters
 - \$* Is the entire list (string) of arguments (excluding the script name)
 - \$# Is the number of parameters (excluding the script name)
 - \$\$ Is the process PID

The **shift** command shifts the parameters to the left (\$0 remains unchanged)

Argument passing example

```
#!/bin/bash
```

```
# Using command line parameters
```

```
echo "Running process is $0"
```

```
echo "Parameters: $1 $2 $3 etc."
```

```
echo "Number of parameters $#"
```

```
echo "List of parameters $@"
```

```
shift
```

```
echo "Parameters: $1 $2 $3 etc."
```

```
shift
```

```
echo "Parameters: $1 $2 $3 etc."
```

```
exit 0
```

The "... " (double quotes) expand the variables

\$0, \$1, etc. can also be written outside "... "

\$0 remains unchanged; consequently \$1=\$2, \$2=\$3, etc.

Again \$0 remains unchanged; consequently \$1=\$2, \$2=\$3, etc.

Variables

❖ Variables can be

➤ **Local** (shell variables)

- Available only in the current shell

➤ **Global** (environment variables)

- Available in all sub-shells
- Are **exported** by the current shell to all the process executed by the shell

Variables

❖ Main features of shell variables

➤ Are not declared

- A variable is created by assigning a value to the variable name

➤ Are case sensitive

- **Var**, **VAR**, and **var** are different variables

➤ Some names are reserved for special purposes

❖ The list of all defined variables and associated value is displayed by command **set**

❖ The **unset** command clears the value of a variable

➤ **unset name**

Local (shell) variables

❖ Characterized by a name and associated content

- The content specifies the type
 - Constant, string, integer, vector or matrix
- The contents associated to a name are strings (even if a string can be interpreted as a numeric value)
- Setting
 - `name="value"`
- Usage
 - `$name`

No blanks around '='

Double quotes are mandatory if the string includes blank characters

Examples

```
> var=Hello
> echo $var
Hello
> var=7+5
> echo $var
7+5
> i="Hello world!"
> echo $i
Hello world!
> i=$i" Bye!!!"
> echo $i
Hello world! Bye!!!
> i=Hello world
> world: command not found
```

Variables are **strings** !!

Strings concatenation

Assign an arithmetic expression to a variable (more details later)

Assignment is incorrect
(do to the blank)
Use quotes

```
> let var=7+5
> echo $var
12
```

Global (environment) variables

- ❖ The **export** command allows creating an environment variable visible by other processes
 - **export name**
- ❖ Notice that
 - Some environment variable names are predefined and reserved
 - When a shell is executed these variables are automatically initialized starting from "environment" values
 - These variable names are typically uppercase
 - Can be displayed by means of the **printenv** (or **env**) command

Example: local and global variable

```
> v=one
> echo $v
one
> bash
> ps -l
... Two bashes running
> echo $v

> exit
> echo $v
one
```

Current shell local variable

This variable is
not set

```
> v=one
> echo $v
one
> export v
> bash
> ps -l
... Two bashes running
> echo $v
one
> exit
> echo $v
one
```

Global variable because it
has been exported by the
sub-shell

Example: variables

Clear video

```
#!/bin/bash
clear
echo "Hello, $USER!"
echo
echo "List logged users"
w #or who
echo "Set two local variables"
COLOR="black"; VALUE="9"
```

w: shows the
logged users

Set commands on
the same line

```
echo "String: $COLOR"
echo "Number: $VALUE"
echo
echo "Completed"
```

```
#exit
```

Also without explicit exit

Partial list

Predefined variables

Variable	Meaning
\$?	Stores the return value of the last process: 0 on success, other than 0 (between 1 and 255) on error. Value 0 corresponds to the TRUE value (unlike in C language)
\$SHELL	Current shell
\$LOGNAME	Username used for login
\$HOME	User home directory
\$PATH	List of the directories, delimited by ':' used for searching the executable files and commands
\$PS1 \$PS2	Main prompt (usually '\$' for users, '#' for root) Auxiliary prompt (usually '>')
\$IFS	Lists the characters that delimits the "words" in an input string (see read shell command)

Examples

```
$ PS1="> "  
> echo $HOME  
...  
> v=$PS1  
> echo $PS1  
...  
> PS1="myPrompt > "  
myPrompt > echo $v  
...
```

shell prompt modifications

```
> myExe  
myExe: command not found  
> PATH=$PATH:.  
> myExe  
... myExe running ...
```

PATH modification,
adding current directory

```
> ls foo  
ls: cannot access foo:  
No such file or directory  
> echo $?  
2  
> ls bar*  
bar.txt  
> echo $?  
0
```

Return value of a
command (0=TRUE)

Read from stdin

- ❖ The **read** function allows reading a line from standard input
- ❖ Syntax
 - **read** [**options**] **var**₁ **var**₂ ... **var**_n
 - **read** can be possibly followed by a list of variables
 - The "words" of the read line will be assigned in turn to each variable
 - Possible excess words are **all** stored (as a string) in the last variable
 - If no variables are specified, the complete input string is stored in variable **REPLY**

Read from stdin

➤ Supported options

- **-n nchars**
 - Returns after reading **nchars** characters without waiting for newline
- **-t timeout**
 - Timeout on reading
 - Returns 1 if a string is not typed within **timeout** seconds
- etc.

Examples: read from stdin

```
> read v
input line string
> echo $v
input line string
```

Input string assigned to
variable **v**

2 variables, but input
string includes 3 words

Input string assigned to
the default variable
REPLY

```
> read v1 v2
input line string
> echo $v1
input
> echo $v2
line string
```

```
> read
> One two three
> echo $REPLY
One two three
> read
One two three
> v=$REPLY
> echo $v
One two three
```

Exercise

- ❖ Write a bash script that takes two integer numbers and prints their sum and product

`-n` no
newline

from stdin

Arithmetic
expression
(more detail
later)

```
#!/bin/bash  
# Sum and product
```

```
echo -n "Reading n1: "  
read n1  
echo -n "Reading n2: "  
read n2  
let s=n1+n2  
let p=n1*n2  
echo "Sum: $s"  
echo "Product: $p"  
  
exit 0
```

No blanks around
`=, +, *`

Exercise

❖ Write a bash script that reads a username, and displays her/his number of logins

➤ The list of logged users is produced by command **who** or **w**

Use of shell commands, variables, etc.

```
#!/bin/bash
# Number of login(s) of a specific user

echo -n "User name: "
read user

# who is logged | look for username | word count
times=$(who | grep $user | wc -l)

echo "User $user has $times login(s)"

exit 0
```

--lines = -l =
of lines

Exercise

- ❖ Write a bash script that reads a string, and displays its length

```
#!/bin/bash
# String length
```

```
echo "Type a word: "
read word
```

echo -n = no new line

```
# echoing without newline | word count chars
l=$(echo -n $word | wc -c)
```

--chars = -c = # of chars
--bytes = -b = # of bytes

```
echo "Word $word is $l characters long"
```

```
exit 0
```


Write to `stdout`

- ❖ Output on `stdout` can be performed using
 - `echo`
 - `printf`
- ❖ Function `printf` syntax is similar to C language `printf`
 - Uses escape characters
 - It is not necessary to delimit fields by `", "`

Write to stdout

❖ echo

- Displays its arguments, delimited by blank, and terminated by newline
- Options
 - -n eliminates the newline
 - -e interprets escaped (\...) characters
 - \b backspace
 - \n newline
 - \t tab
 - \\ backslash
 - etc.

Examples: I/O

```
echo "Printing with a newline"
echo -n "Printing without newline"
echo -e "Deal with \n escape \t\t characters"
printf "Printing without newline"
printf "%s \t%s\n" "Hello. It's me:" "$HOME"
```

Output: Hello. It's me: /home/scanzio

I & O together inside the same script

```
#!/bin/bash
# Interactive input/output
echo -n "Insert a sentence: "
read w1 w2 others
echo "Word 1 is: $w1"
echo "Word 2 is: $w2"
echo "The rest of the line is: $others"
exit 0
```

Arithmetic expressions

- ❖ Several notations can be used for defining arithmetic expressions
 - Command `let "..."`
 - Double parentheses `((...))`
 - Square parentheses `[...]`
 - Syntactic statement **expr**
 - Evaluates an expression by means of a new shell
 - Less efficient
 - Normally not used

Notice that an arithmetic expression is evaluated as TRUE
(exit status) IFF it is not 0
expression !=0 → FALSE exit status=0 → TRUE

Examples

Alternative syntaxes for arithmetic expressions

Use of `((e))`

```
> i=1
> ((v1=i+1))
> ((v2=$i+1))
> v3=$(( $i+1 ))
> v4=$(( i+1 ))
> echo $i $v1 $v2 $v3 $v4
1 2 2 2 2
```

Use of `let`

```
> i=1
> let v1=i+1
> let "v2 = i + 1"
> let v3=$i+1
> echo $i $v1 $v2 $v3
1 2 2 2
```

Use of `[e]`

```
> i=1
> v1=$(( $i+1 ))
> v2=$(( i+1 ))
> echo $i $v1 $v2
1 2 2
```

If it is not between `"..."` the expression **cannot** include blanks

Conditional statement: if-then-fi

- ❖ The conditional statement **if-then-fi**
 - Checks if the exit status of a sequence of commands is equal to 0
 - Recall: 0=TRUE in UNIX shell
 - If so, it executes one or more commands
- ❖ The statement can also include an else condition statement
 - **if-then-else-fi**
 - which allows also performing nested checks
 - **if-then-...-if-then-...-fi-fi**
 - **if-then-elif-...-fi**

Conditional statement: if-then-fi

```
# Syntax 1
if condExpr
then
    statements
fi
```

Standard
format

Statement on a
single line: ';' is mandatory

```
# Syntax 2
if condExpr ; then
    statements
fi
```

With else

```
# Syntax 3
if condExpr
then
    statements
else
    statements
fi
```

Nested
if-then-else-fi
can be written as
if-then-elif-fi

```
# Syntax 4
if condExpr
then
    statements
elif condExpr
then
    statements
else
    statements
fi
```

Conditional statement: if-then-fi

❖ condExpr

- Conditional expressions can use two syntactic flavors

Syntax 1
test param op param

Different operators for

- Numbers
- Strings
- Logical values
- Files and directories

Syntax 2
[param op param]

Square parentheses must be
delimited by a blank

Conditional statement: if-then-fi

Operators for numbers

-eq	==
-ne	!=
-gt	>
-ge	>=
-lt	<
-le	<=
!	! (not)

Operators for files and directories

-d	Argument is a directory
-f	Argument is a regular file
-e	Argument exists
-r	Argument has read permission
-w	Argument has write permission
-x	Argument has execution permission
-s	Argument has non-null dimension

Operators for strings

=	strcmp
!=	strcmp
-n string	non NULL string
-z string	NULL (empty) string

Logical operators

!	NOT
-a	AND (inside [])
-o	OR (inside [])
&&	AND (in a sequence of commands)
	OR (in a sequence of commands)

Examples

```
if [ 0 ]      # true, interpreted as a string
if [ 1 ]      # true
if [ -1 ]     # true
if [ ]        # NULL is false
if [ str ]    # a random string is true,
               # e.g., "abc" or abc is true
```

Logical values

Test on numbers

```
if [ $v1 -eq $v2 ]
then
    echo "v1==v2"
fi
```

or

```
if test $v1 -eq $v2
```

```
if [ $v1 -lt 10 ]
then
    echo "$v1 < 10"
else
    echo "$v1 >= 10"
fi
```

Examples: file check

```
if [ "$a" -eq 24 -a "$s" = "str" ]; then
    ...
fi
```

AND of conditions

Equivalent format ([\equiv test command)

```
if [ "$a" -eq 24 ] && [ "$s" = "str" ]
if [[ "$a" -eq 24 && "$s" = "str" ]]
```

```
if [ $recursiveSearch -eq 1 -a -d $2 ]
then
    find $2 -name *.c > $3
else
    find $2 -maxdepth 1 *.c > $3
fi
```

Examples: string check

```
if [ $string = "abc" ]; then  
    echo "string \"abc\" found"  
fi
```

Test on strings

If \$string is null (e.g., return from input) the syntax is incorrect because is evaluated as: `[= "abc"]`

Use double quotes for a error resistant syntax:

```
if [ "$string" = "abc" ]; then  
    which would be evaluated as: [ "" = "abc" ]
```

```
if [ -f foo.c ]; then  
    echo "foo.c is in this directory"  
fi
```

Test on file

Examples: whole script

```
#!/bin/sh
```

```
echo -n "Is it morning (yes/no)? "  
read string  
if [ "$string" = "yes" ]; then  
    echo "Good morning"  
else  
    echo "Good afternoon"  
fi  
  
exit 0
```

Reading string from stdin
Check the string
Display of the output

Examples: whole script

Reading string from stdin
Check the string
Display of the output
Use of elif

```
#!/bin/sh
```

```
echo -n "Is it morning (yes/no)? "  
read string  
if [ "$string" = "yes" ]; then  
    echo "Good morning"  
elif [ "$string" = "no" ]; then  
    echo "Good afternoon"  
else  
    echo "Sorry, wrong answer"  
fi  
  
exit 0
```

Iterative statement for-in

❖ Statement **for-in** (**for var in list**)

- Executes the commands, for each value taken by variable **var**
- The list of values can be given
 - Explicitly (**list**)
 - Implicitly (result of shell commands di shell, wild-cards, etc.)

```
# Syntax 1
for var in list
do
    statements
done
```

```
# Syntax 2
for var in list; do
    statements
done
```

Remark: definite construct, i.e., iterates a **predefined** number of times

Examples: for with list

```
for foo in 1 2 3 4 5 6 7 8 9 10
do
    echo $foo
done
```

Displays a list of
"numbers"

```
for str in foo bar echo charlie tango
do
    echo $str
done
```

Displays a list of strings

```
num="2 4 6 9 2.3 5.9"
for file in $num
do
    echo $file
done
```

Displays a list of numbers
using a variable
(arrays, see later)

Examples: for and wild-chars

```
n=1
```

```
for i in $* ; do
```

```
    echo "par #" $n = $i
```

```
    let n=n+1
```

```
done
```

Iterates on the parameter
of the scripts

Displays all the parameters
received on the command line

```
for f in $(ls | grep txt); do
```

```
    chmod g+x $f
```

```
done
```

Change privileges to
specific files

```
rm -f number.txt
```

```
for i in $(echo {1..50})
```

```
do
```

```
    echo -n "$i " >> number.txt
```

```
done
```

Append the numbers from
1 to 50 to file
number.txt, in the same
line and separated by a space

'>' would overwrite
number.txt at every
iteration

Examples: for and wild-chars

```
for file in [ab]* ; do  
    rm -fr $file  
    echo "Removing file $file"  
done
```

Remove files with name
beginning by
a OR b

Changes the
privileges of files with
name including digit 7

```
for f in $(ls | grep 7); do chmod g+x $f; done
```


Iterative statement while-do-done

- ❖ Iterates while the condition is true
 - the number of iterations is unknown

Syntax 1

```
while [ cond ]  
do  
    statements  
done
```

Syntax 2

```
while [ cond ] ; do  
    statements  
done
```

Example

```
#!/bin/bash
```

```
limit=10
```

```
var=0
```

```
while [ "$var" -lt "$limit" ]
```

```
do
```

```
    echo "Here var is equal to $var"
```

```
    let var=var+1
```

```
done
```

```
exit 0
```

Displays 10 times a message

Example

```
#!/bin/bash
```

```
echo "Enter password: "
```

```
read myPass
```

```
while [ "$myPass" != "secret" ]; do
```

```
    echo "Sorry. Try again."
```

```
    read myPass
```

```
done
```

```
exit 0
```

Displays a message
until the correct
string is given

Example of read with stdin redirection

```
#!/bin/bash
```

```
n=1
```

```
while read row
```

```
do
```

```
    echo "Row $n: $row"
```

```
    let n=n+1
```

```
done < in.txt > out.txt
```

```
exit 0
```

Reads complete lines from stdin

Writing
echo ... > out.txt
implies to rewrite file out.txt at any iteration. You can use:
echo ... >> out.txt

Writing
while read row < in.txt
will always re-read the first line of the file

Constant filenames.
Possibility to use parameters or variables: ... **<\$1 > \$var**

Since the while-do-done statement is considered to be unique, the redirection (of I/O) must be done at the end of the statement

Exercise

- ❖ Write a bash script that
 - Takes two integers **n1** and **n2** from command line, otherwise reads them from **stdin** (if not present)
 - Display a matrix of **n1** rows and **n2** columns of increasing integer values starting from 0
 - Example

```
> ./myScript 3 4
```

```
0  1  2  3
```

```
4  5  6  7
```

```
8  9 10 11
```

Solution

```
#!/bin/bash
if [ $# -lt 2 ] ; then
    echo -n "Values: "
    read n1 n2
else
    n1=$1
    n2=$2
fi
```

Reads input
data

```
n=0
r=0
while [ $r -lt $n1 ] ; do
    c=0
    while [ $c -lt $n2 ] ; do
        echo -n "$n "
        let n=n+1
        let c=c+1
    done
    let r=r+1
    echo
done
exit 0
```

Double loop
for displaying
the values

Break, continue and ':'

- ❖ **break** and **continue** statements have the same meaning in shell and in C language
 - **break**: unstructured exit from the cycle
 - **continue**: skip to the next iteration of the cycle
- ❖ Character **:** can be used
 - For creating "null instructions"
 - `if [-d "$file"]; then`
 - `: # Empty instruction`
 - `fi`
 - For indicating a TRUE condition
 - `while :`
 - equivalent to `while [0]`

Arrays

❖ **bash** define also one-dimensional arrays

- Any variable can be defined as an array
 - Explicit declaration is not required (but possible with the **declare** construct)
- No restriction
 - On the dimension of the array
 - On the use of contiguous indices
- Indices usually start from 0
 - Zero-base indexing, as in C language

Arrays in shell are **not** associative (no hashing)

Arrays

❖ Suppose **name** is the name of a vector

➤ Definition

- Element-wise
 - `name[index]="value"`
- By means of a list of values
 - `name = (list of values separated by blanks)`

A new element can be created at any time

➤ Reference

- A single element
 - `${name[index]}`
- All elements
 - `${name[*]}`

The use of `{ }` is mandatory

* or @

Arrays

- Number of elements

- `${#name[*]}`

- Length of the i-th element (number of characters)

- `${#name[i]}`

- ❖ Statement **unset** eliminates

- an element

- `unset name[index]`

- an array

- `unset name`

Examples: arrays

Initialized by a list

```
> vet=(1 2 5 hello)
> echo ${vet[0]}
1
> echo ${vet[*]}
1 2 5 hello
> echo ${vet[1-2]}
2 5
> vet[4]=bye
> echo ${vet[*]}
1 2 5 hello bye
```

Elimination

```
> unset vet[0]
> echo ${vet[*]}
2 5 hello bye
> unset vet
> echo ${vet[*]}

> vet[5]=100
> vet[10]=50
> echo ${var[*]}
100 50
```

Non contiguous
indexes

Exercise

- ❖ Write a bash script that
 - Reads a sequence of numbers, one per line, ending by 0
 - Displays the values read in inverse order
 - Example

Input n1: 14

...

Input n10: 123

Input n11: 0

Output: 123 ... 14

Solution

```
#!/bin/bash
```

or :

```
i=0
```

```
while [ 0 ]; do
```

```
    echo -n "Input $i: "
```

```
    read v
```

```
    if [ "$v" -eq "0" ] ; then
```

```
        break;
```

```
    fi
```

```
    vet[$i]=$v
```

```
    let i=i+1
```

```
done
```

Input

echo \$ {vet [*]}
would display the
elements in the same
order and separated
by a space

```
echo
```

```
let i=i-1
```

```
while [ "$i" -ge "0" ]
```

```
do
```

```
    echo "Output $i: ${vet[$i]}"
```

```
    let i=i-1
```

```
done
```

```
exit 0
```

Output
in inverse order