

- History
- Recommended sources to help
- Variables
 - Дефолтные переменные
- Bash options
 - set
 - shopt
- Коды завершения
- Производительность.
- Начало скрипта
- Кавычки
- Типы данных
- Пользовательский ввод
- Ссылки на файлы с переменными и функциями
- Аргументы скрипта
 - Positional arguments
 - More complex arguments
- Оператор shift
- Command substitution
- Parameter substitution
- Here document
- Function
 - Arguments
 - Variable scope
- Simple script example
- Pattern matching
 - почистить строку
 - сделать замену в переменной
 - Поменять регистр
 - примеры скриптов
- Примеры использования ext globing
- Численные вычисления
 - Операции с целыми числами
 - Операции с дробными числами
 - bc - утилита для дробных вычислений
 - Полезные утилиты для вычислений
 - Примеры скриптов
- Ветвление
 - if
 - case
- Циклы
 - for
 - while
 - until (opposite while)
 - break and continue
- Menu

- [trap](#)
- [Arrays](#)
 - [Types of arrays:](#)
 - [Reference to array values](#)
 - [Example script indexed array](#)
 - [Example of associative array](#)
 - [Reading output to an array](#)
 - [Looping on array](#)
 - [Lab about arrays](#)
- [Best Practises](#)
 - [Using BASH_XTRACEFD for debug](#)
- [Complex scripts](#)
 - [trivial restart if service down](#)
 - [check users are created](#)
 - [install kube cluster](#)

History

Understanding Shell History

- The Bash shell goes back to shells that were created for use in UNIX in the 1970s
- Bourne shell (/bin/sh) was the original shell
- C-shell (/bin/csh) was developed as a shell that is very close to the C programming language
- Korn shell (/bin/ksh) was created as a shell that offers the best of Bourne and C-shell

Understanding Linux Shells

- Bash is Bourne Again Shell, a remake of the original Bourne shell that was invented in the early 1970's
- Bash is the default shell on most Linux distributions
- All other common Linux shells are a fork of Bourne shell
- Another common shell is Zsh, which is used as the default shell on MacOS
- And yet another common shell is Dash, which is used frequently in Debian environments
- While writing shell scripts, Bash is the standard and it's very easy to make Bash work, even if you're in a non-Bash shell

Bash это Bourne again shell

Recommended sources to help

- `man bash`
- [Text book for Beginner. Recommended by Sander van Vugt](#)
- [Text book for Advanced. Recommended by Sander van Vugt](#)

Variables

When you create new variable it can be seen only in current bash. Many commands invoke another bash. In that case you can use `export`

```
vagrant@CONTROL-VM:~$ myvar=1
vagrant@CONTROL-VM:~$ echo $myvar
1
vagrant@CONTROL-VM:~$ bash
vagrant@CONTROL-VM:~$ echo $myvar

vagrant@CONTROL-VM:~$ exit
exit
vagrant@CONTROL-VM:~$ echo $myvar
1
vagrant@CONTROL-VM:~$
```

we can create variable and at the same place export to all child bashes

```
export KEY=VALUE
```

Variables is case-insensitive

to make variable null again

```
variable=
```

delete variable

```
unset variable
```

Ссылка на переменную:

- `$variable`
- `${variable}` рекомендуемый способ
- `echo "${variable}"` в некоторых случаях предпочтительно использовать ссылку внутри двойных кавычек

Дефолтные переменные

`$RANDOM` - случайное число

`$SECONDS` - число секунд, которое работает текущий shell

`$LINENO` - номер строки текущего скрипта

`$HISTCMD`- номер текущей команды в истории команд

`$GROUPS`- массив с именами групп, в которые входит текущий пользователь

`$DIRSTACK`- история недавно посещаемых директорий

`$BASH_ENV` `$BASH_OPT` -

Bash options

посмотреть все опции bash

```
man bash
```

builtin command, including `-o`, can be used as options when the shell is invoked. In addition, bash interprets the following options when it is invoked:

<code>-c</code>	If the <code>-c</code> option is present, then commands are read from the first non-option argument <u>command string</u> . If there are arguments after the <u>command string</u> , the first argument is assigned to <code>\$0</code> and any remaining arguments are assigned to the positional parameters. The assignment to <code>\$0</code> sets the name of the shell, which is used in warning and error messages.
<code>-i</code>	If the <code>-i</code> option is present, the shell is <u>interactive</u> .
<code>-l</code>	Make bash act as if it had been invoked as a login shell (see INVOCATION below).
<code>-r</code>	If the <code>-r</code> option is present, the shell becomes <u>restricted</u> (see RESTRICTED SHELL below).
<code>-s</code>	If the <code>-s</code> option is present, or if no arguments remain after option processing, then commands are read from the standard input. This option allows the positional parameters to be set when invoking an interactive shell or when reading input through a pipe.
<code>-v</code>	Print shell input lines as they are read.
<code>-x</code>	Print commands and their arguments as they are executed.
<code>-D</code>	A list of all double-quoted strings preceded by <code>\$</code> is printed on the standard output. These are the strings that are subject to language translation when the current locale is not C or POSIX. This implies the

Manual page bash(1) line 23 (press h for help or q to quit)

set

Меняют поведение программ. Например `-x` меняет работу команд, при старте команды будет выводиться в stdout сама команда и ее аргументы. Удобно для дебага скриптов.

```
set -x
ls
set +x
```

```
dima@LAPTOP-MIH0IT98:/mnt/c/Users/anton/NOTES/personal-notes/diary$ ls
diary.md
dima@LAPTOP-MIH0IT98:/mnt/c/Users/anton/NOTES/personal-notes/diary$ set -x
dima@LAPTOP-MIH0IT98:/mnt/c/Users/anton/NOTES/personal-notes/diary$ ls
+ ls --color=auto
diary.md
dima@LAPTOP-MIH0IT98:/mnt/c/Users/anton/NOTES/personal-notes/diary$ set +x
+ set +x
dima@LAPTOP-MIH0IT98:/mnt/c/Users/anton/NOTES/personal-notes/diary$ ls
diary.md
dima@LAPTOP-MIH0IT98:/mnt/c/Users/anton/NOTES/personal-notes/diary$ █
```

Варианты использования:

- поставить в текущей консоли для временного использования (удобно для дебага) `bash -x ./my_script`
- поставить в скрипты для пользователя или пользователей при старте
- включать для действия в конкретном скрипте через `#!/bin/bash -x`
- `-e` - exit the script when a command fail

- `-i` - run script in interactive mode - line-by-line
- `-v` - verbose mode. A bit less details then `-x`

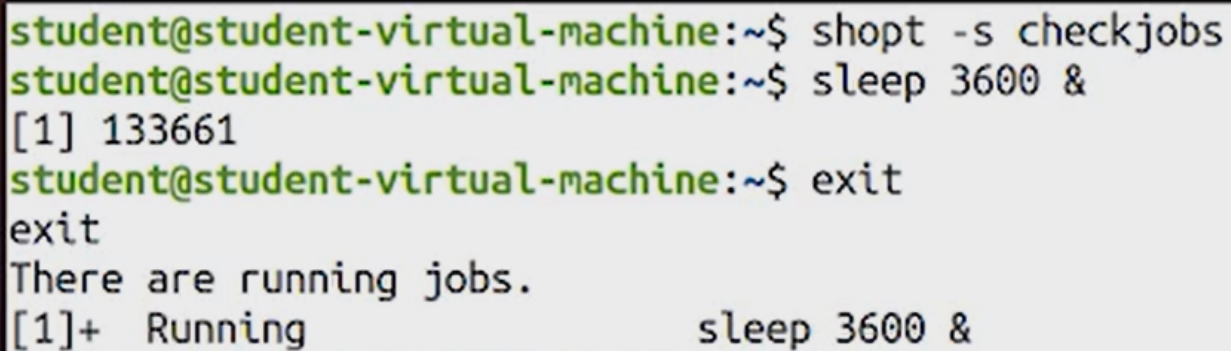
shopt

Некоторые опции ставятся через команду `shopt`. Они также описаны в `man bash`

показать все опции `shopt`

пример включения мониторинга джобов при выходе

```
shopt -s checkjobs  
sleep 3600 &  
exit
```



```
student@student-virtual-machine:~$ shopt -s checkjobs  
student@student-virtual-machine:~$ sleep 3600 &  
[1] 133661  
student@student-virtual-machine:~$ exit  
exit  
There are running jobs.  
[1]+  Running                  sleep 3600 &
```

Коды завершения

каждая команда заканчивается каким-то кодом

посмотреть предыдущий код

```
echo $?
```

0 - успешно 1 - ошибка общего характера general error Обычно все что не 0 это какая-то проблема.

скрипт можно завершить кодом

```
exit n где n это код
```

Производительность.

- Для лучшей производительности лучше не использовать утилиты, подгружаемые с диска, например `sed` и `awk`.
- По возможности использовать встроенными в баш командами.
- Команда `help` выводит все встроенные команды в баш.
- Команда `type` выводит тип команды.
- Командой `time` можно проверить время исполнения конкретной команды.
- Можно посмотреть от каких библиотек зависит команда, командой `ldd`. Подробнее смотри заметки по линуксу.

Начало скрипта

Для надежности стоит писать в самом начале скрипта

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

или

```
#!/usr/bin/env bash
```

в этом случае если текущий shell не `bash`, например `zsh`, все равно будет исполняться `bash`

проверить текущий shell

```
echo $SHELL
```

Кавычки

" - двойные кавычки используются для указания того, что проблемы в подстроке относятся к одному аргументу. Все внедрения переменных через `$` будут работать.

' - одинарные кавычки используются для остановки внедрения переменных и скриптов.

Типы данных

В баше нет типов данных, но есть массивы. Также есть ключевое слово `declare`, которое используется для указания некоторых свойств переменных

Установка свойства `read-only` на переменную `ANSWER` со значением `yes`

```
declare -r ANSWER=yes
```

указание что переменная является индексированным или ассоциативным массивом

```
declare -a MYARRAY
```

печатает тип переменной

```
declare -p VARIABLE
```

Пользовательский ввод

самый простой способ

```
echo enter the value
read value
echo you have entered $value
```

если после `read` не указано переменной, ввод сохраняется в переменной `$REPLY`

удобный способ завершать скрипт с фразой `Press any key to continue`

```
echo Press Enter to continue
read
```

можно использовать для установки нескольких переменных

```
echo enter firstname, lastname and city
read firstname lastname city
echo nice to meet you $firstname $lastname from $city
```

Ссылки на файлы с переменными и функциями

Есть механизм подключения файлов в текущий `bash` это называется `sourcing`

два способа, после начала скрипта написать:

```
source path_to_file_with_variables
```

```
. path_to_file_with_variables
```

Аргументы скрипта

Positional arguments

Можно ссылаться на аргументы скрипта через `$1` и до `$9`. Если аргументов больше тогда ссылки будут с использованием фигурных скобок `${10}`.

Переменная `$0` содержит имя текущего скрипта.

Переменная `$#` содержит количество аргументов

Также можно ссылаться сразу на все аргументы через `$@` и `$*`. Без кавычек эти способы равнозначны. С кавычками `$@` массив аргументов по которому можно итерироваться `$*` с кавычками помещает все аргументы в один аргумент. Рекомендуется использовать `$@` во всех случаях, если не нужно поместить все аргументы в массив.

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

echo "Hello $1, how are you today"
echo " hello $2, how are you"
echo " hello $10, how are you"
echo " hello ${10}"
echo " hello ${11}"
shift
echo hi $1
echo "\$0 is $0"
```

```
student@student-virtual-machine:~/bash-scripting$ ./script3 a b c d e f g h i j k l
Hello a, how are you today
hello b, how are you
hello a0, how are you
hello j
hello k
hi b
$0 is ./script3
```

```
#!/bin/bash
echo "Hello $1, how are you today"
echo "\$# gives $#"
```

```
# trying to show every single argument on a separated line
echo showing the interpretation of \*$*
for i in "$*"
do
    echo $i
done

echo showing the interpretation of \*$@
for i in "$@"
do
    echo $i
done
```

```
student@student-virtual-machine:~/bash-scripting$ ./script4 a b c d
Hello a, how are you today
$# gives 4
$* gives a b c d
$@ gives a b c d
$0 is ./script4
showing the interpretation of $*
a b c d
showing the interpretation of $@
a
b
c
d
```

More complex arguments

```
#!/bin/bash
while getopts "hs:" arg; do
case $arg in
    h)
        echo "usage"
        ;;
    s)
        strength=$OPTARG
        echo $strength
        ;;
    esac
done
```

```

student@student-virtual-machine:~/bash-scripting$ vim options
student@student-virtual-machine:~/bash-scripting$ ./options -h
usage
student@student-virtual-machine:~/bash-scripting$ ./options -s 5
5
student@student-virtual-machine:~/bash-scripting$ ./options -s
./options: option requires an argument -- s
student@student-virtual-machine:~/bash-scripting$ ./options -x
./options: illegal option -- x

```

complex example

```

#!/bin/bash
#makeusr [-u uid] [-g gid] [-i info] [-h homedir] [-s shell] username
function usage
{
    echo 'usage: makeusr [-u uid] [-g gid] [-i info] [-h homedir] '
    echo '[-s shell] username'
    exit 1
}

function helpmessage
{
    echo "makeusr is a script ..."
    echo "blablabla"
}

while getopts "u:g:i:h:s:" opt; do
    case $opt in
        u ) uid=$OPTARG ;;
        g ) gid=$OPTARG ;;
        i ) info=$OPTARG ;;
        h ) home=$OPTARG ;;
        s ) shell=$OPTARG ;;
        ? ) helpmessage ;;
        * ) usage ;;
    esac
done
shift $(( $OPTIND - 1 ))

if [ -z "$1" ]; then
    usage
fi

if [ -n "$2" ]; then
    usage
fi

if [ -z "$uid" ]; then
    uid=500

```

```

        while cut -d : -f3 /etc/passwd | grep -x $uid
        do
            uid=$((uid+1)) > /dev/null
        done
    fi

    if [ -z "$gid" ]; then
        gid=$(grep users /etc/group | cut -d: -f3)
    fi

    if [ -z "$info" ]; then
        echo Provide information about the user.
        read info
    fi

    if [ -z "$home" ]; then
        home=/home/$1
    fi

    if [ -z "$shell" ]; then
        shell=/bin/bash
    fi

    echo $1:x:$uid:$gid:$info:$home:$shell >> /etc/passwd
    echo $1::::::::: >> /etc/shadow
    mkdir -p $home
    chmod 660 $home
    chown $1:users $home
    passwd $1

```

manual argument processing

```

#!/usr/bin/env bash

# File name
readonly PROGNAME=$(basename $0)
# File name, without the extension
readonly PROGBASENAME=${PROGNAME%. *}
# File directory
readonly PROGDIR=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)
# Arguments
readonly ARGS="$@"
# Arguments number
readonly ARGNUM="$#"

usage() {
    echo "Script description"
    echo
    echo "Usage: $PROGNAME -i <file> -o <file> [options]..."
    echo
    echo "Options:"

```

```

echo
echo "  -h, --help"
echo "      This help text."
echo
echo "  -i <file>, --input <file>"
echo "      Input file. If \"-\", stdin will be used instead."
echo
echo "  -o <file>, --output <file>"
echo "      Output file."
echo
echo "  --"
echo "      Do not interpret any more arguments as options."
echo
}

while [ "$#" -gt 0 ]
do
    case "$1" in
        -h|--help)
            usage
            exit 0
            ;;
        -i|--input)
            input="$2"

            # Jump over <file>, in case "-" is a valid input file
            # (keyword to standard input). Jumping here prevents reaching
            # "-*)" case when parsing <file>
            shift
            ;;
        -o|--output)
            output="$2"
            ;;
        --)
            break
            ;;
        -*)
            echo "Invalid option '$1'. Use --help to see the valid options" >&2
            exit 1
            ;;
        # an option argument, continue
        *) ;;
    esac
    shift
done

```

Оператор shift

Удаляет аргументы скрипта при использовании.

Если не указывать ничего после `shift` сместит все аргументы влево, т.е первый аргумент будет удален, а второй займет его место и так далее.

Если указать цифру после `shift` в этом случае произойдет смещение влево аргументов на величину цифры

```
#!/bin/bash
echo the script has $# arguments

echo print $1
shift
echo print $1
shift 3
echo print $1
```

```
student@student-virtual-machine:~/bash-scripting$ ./myshift a b c d e f g h i j k
the script has 11 arguments
print a
print b
print e
```

Command substitution

Two cases

- one

```
today=$(date +%d-%m-%y)
mykernel=$(uname -r)
```

- two

```
today=`date +%d-%m-%y`
mykernel=`uname -r`
```

for readability use this `$()`

```
#!/bin/bash
#simple demo backup script
#writes backup file to current directory and backs up everything provided
as an argument

if [ -z $1 ]
then
    echo argument required
```

```
exit 9

fi

sudo tar -cvf $(date +%d-%m-%y).bak $@
```

Parameter substitution

можно ставить дефолтное значение в случае отсутствия значения, либо вызывать выражение, которое определит значение

```
#!/bin/bash

echo press username or press enter to use default value
read username
echo ${username:-$(whoami)}
```

тут выполняется попытка чтения инпута в переменную username. Далее через `${}` осуществляется ссылка на переменную, если она пустая, то вызывается command substitution `whoami` который возвращает имя текущего пользователя. При этом присвоение к username не происходит, оно по-прежнему пустое.

Если `$1` не задан, то используется дефолтное значение и присваивается к `filename`

```
filename=${1:-$DEFAULT_FILENAME}
```

тут происходит присваивание в случае отсутствия

```
echo ${username:=$(whoami)}
filename=${1:=DEFAULT_FILENAME}
```

```
#!/bin/bash
echo take one
echo ${var:-abc}
echo ${var}

echo take two
echo ${var:=abc}
echo ${var}
```

если переменная не задана пишем сообщение и выходим с кодом 1

```
echo ${myvar:?error_message}
```

Here document

A here document is a special-purpose code block. It uses a form of I/O redirection to feed a command list to an interactive program or a command, such as ftp, cat, or the ex text editor.

```
COMMAND <<InputComesFromHERE
...
...
...
InputComesFromHERE
```

```
ssh $SERVER bash <<EOF
cd downloads/
read -e -p "Enter the path to the file: " FILEPATH
echo $FILEPATH
eval FILEPATH="$FILEPATH"

echo "Downloading $FILEPATH to $CLIENT"
EOF
```

```
cat << REBOOT >> /root/completeme.sh
touch /tmp/after-reboot
rm -f /etc/profile
mv /etc/profile.bak /etc/profile
echo DONE
REBOOT

chmod +x /root/completeme.sh
cp /etc/profile/ etc/profile.bak
echo /etc/completeme.sh >> /etc/profile

reboot
```

Function

Two option to write function

spaces between braces are not matter


```
function_name () {  
  
}
```

```
function function {  
  
}
```

Arguments

Function argument has local scope

```
#!/bin/bash  
hello(){  
    echo hello $1  
}  
hello bob
```

Arguments for all script are unavailable for function

Variable scope

No matter where they are defined, variables always have a global scope - even if defined in a function

`local` keyword to define local variable

```
#!/bin/bash  
var1=A  
  
my_function () {  
    local var2=B  
    var3=C  
    echo "inside function: var1: $var1, var2: $var2, var3: $var3"  
}  
  
echo "before running function: var1: $var1, var2: $var2, var3: $var3"  
  
my_function  
  
echo "after running function: var1: $var1, var2: $var2, var3: $var3"
```

```
student@student-virtual-machine:~/bash-scripting$ ./funcvar  
before running function: var1: A, var2: , var3:  
inside function: var1: A, var2: B, var3: C  
after running function: var1: A, var2: , var3: C
```

Simple script example

```
#!/bin/bash

if grep -i 'ubuntu' /etc/os-release > /dev/null
then
    PACKAGE_MANAGER=apt
fi

if grep -i 'red hat' /etc/os-release > /dev/null
then
    PACKAGE_MANAGER=yum
fi

if [ -z $1 ]
then
    echo enter package name to install
    read PACKAGE
else
    PACKAGE=$1
fi

sudo $PACKAGE_MANAGER install $PACKAGE -y
```

Pattern matching

Рекомендуется к использованию, а не `sed` т.к это нативный `bash`, ничего с диска не читается, `sed` - утилита на диске.

длинну аргумента. Количество символов.

```
${1#}
```

почистить строку

- `${var#pattern}` удаляет самый короткий результат поиска слева направо
- `${var##pattern}` удаляет самый длинный результат поиска слева направо
- `${var%pattern}` удаляет самый короткий результат поиска справа налево
- `${var%%pattern}` удаляет самый длинный результат поиска справа налево

```
#!/bin/bash
#
# ...
# to test, use /usr/bin/blah
```

```
filename=${1#*/}  
echo 'filename=${1#*/}'  
echo "The name of the file is $filename"  
directoryname=${1%/*}  
echo 'directoryname=${1%/*}'  
echo "the name of the directory is $directoryname"
```

```
student@student-virtual-machine:~/bash-scripting$ ./script6 /usr/bin/blah  
filename=${1#*/}  
The name of the file is blah  
directoryname=${1%/*}  
the name of the directory is /usr/bin
```

```
#!/bin/bash  
BLAH=rababarabarabara  
clear  
  
echo BLAH=$BLAH  
echo 'the result of ##*ba is' ${BLAH##*ba}  
echo 'the result of #*ba is' ${BLAH#*ba}  
echo 'the result of %%ba* is' ${BLAH%%ba*}  
echo 'the result of %ba* is' ${BLAH%ba*}
```

```
BLAH=rababarabarabarabara  
the result of ##*ba is rara  
the result of #*ba is barabarabarabara  
the result of %%ba* is ra  
the result of %ba* is rababarabara
```

сделать замену в переменной

```
${var/pattern/replacement}
```

да очень похоже на `sed`

для глобальной замены, т.е много раз используем двойной слеш

```
${var//pattern/replacement}
```

делать замену, только если переменная начинается с паттерна

```
${var/#pattern/replacement}
```

делать замену, только если переменная заканчивается паттерном

```
${var/%pattern/replacement}
```

```
#!/bin/bash

VAR=donkey
echo $VAR

VAR=${VAR/donkey/horse}
echo $VAR
```

Поменять регистр

- `${var^^}` переводит в верхний регистр
- `${var,,}` переводит в нижний регистр

```
#!/bin/bash

color=red
echo ${color^^}

color=BLUE
echo ${color,,}
```

примеры скриптов

Заменить в текущей директории названия файлов с .txt на название без расширения

```
#!/bin/bash

for i in *.txt
do
    mv $i ${i%.*}
done
```

Примеры использования ext globbing

Итерируемся по файлам в текущей директории, пишем имя файла в переменную `i`. В случае если выполняется выражение, т.е файл не заканчивается на doc, txt, pdf выводится строка с именем файла, что это не документ, иначе выводится строка, что этот файл документ.

```
#!/bin/bash
shopt -s extglob
for i in *
do
    case $i in
        !(*.doc|*.txt|*.pdf))
            echo $i is not a document
            ;;
        *)
            echo $i is a document
            ;;
    esac
done
```

Итерируемся по файлам в текущей директории и выводим их список, но удаляем расширения определенные.

```
#!/bin/bash
shopt -s extglob
for i in *
do
    echo ${i%*(*.doc|*.txt|*.pdf)}
done
```

чуть подправим, чтобы не показывал, а прямо переименовывал файлы

```
#!/bin/bash
shopt -s extglob
for i in *
do
    mv $i ${i%*(*.doc|*.txt|*.pdf)}
done
```

Численные вычисления

Операции с целыми числами

Несколько способов посчитать целочисленное выражение

- `let expression` встроенное в баш
- `expr expression` внешняя программа - редко используется.

- `$((expression))` встроенное в баш. Рекомендуемый способ

Операторы: `+`, `++`, `-`, `%`, `*`, `/`

```
let a=1+2
echo $a
let a++
echo $a

echo $(( 1 * 3 ))
```

что-нибудь посложнее

```
echo $(( $(sudo fdisk -l | grep '/dev/sda:' | awk '{ print $5 }') / 1024 ))
echo $(( $(sudo fdisk -l | grep '/dev/sda:' | awk '{ print $5 }') / $(
1024 * 1024 )) ))
```

Операции с дробными числами

bc - утилита для дробных вычислений

К сожалению ее нету в gitbash Обычно используется в pipes

```
echo "12/5" | bc
```

для того чтобы в результате присутствовала дробная часть используется опция `-l`

```
echo "12/5" | bc -l
```

Поддерживает разные математические функции, например

```
echo "sqrt(1000)" | bc -l
```

Полезные утилиты для вычислений

разложение на простые множители `factor`

```
factor 3444
```

output: 3444: 2 2 3 7 41

Примеры скриптов

Заметим что **COUNTER** в выражении понимается как переменная и не требует ссылки через **\$**

```
#!/bin/bash

COUNTER=$1
COUNTER=$(( COUNTER * 60 ))

while true
do
    echo $COUNTER seconds remaining in break
    COUNTER=$(( COUNTER - 1 ))
    sleep 1
done
```

Ветвление

if

внутри выражение обычно используются модификации команды **test**. О ней можно почитать в

```
man test
```

DESCRIPTION

Exit with the status determined by EXPRESSION.

--help display this help and exit

--version
output version information and exit

An omitted EXPRESSION defaults to false. Otherwise, EXPRESSION is true or false and sets exit status. It is one of:

(EXPRESSION)
EXPRESSION is true

! EXPRESSION
EXPRESSION is false

EXPRESSION1 **-a** EXPRESSION2
both EXPRESSION1 and EXPRESSION2 are true

EXPRESSION1 **-o** EXPRESSION2
either EXPRESSION1 or EXPRESSION2 is true

-n STRING
the length of STRING is nonzero

STRING equivalent to **-n** STRING

-z STRING
the length of STRING is zero

вместе с тем что есть test как внешняя команда, есть test, который является внутренней командой. И получить помощь по ней можно как обычно по встроенным командам в баш `help test`


```
antonov@antonov-VirtualBox:~/Desktop$ help test
```

```
test: test [expr]
```

Evaluate conditional expression.

Exits with a status of 0 (true) or 1 (false) depending on the evaluation of EXPR. Expressions may be unary or binary. Unary expressions are often used to examine the status of a file. There are string operators and numeric comparison operators as well.

The behavior of test depends on the number of arguments. Read the bash manual page for the complete specification.

File operators:

-a FILE	True if file exists.
-b FILE	True if file is block special.
-c FILE	True if file is character special.
-d FILE	True if file is a directory.
-e FILE	True if file exists.
-f FILE	True if file exists and is a regular file.
-g FILE	True if file is set-group-id.
-h FILE	True if file is a symbolic link.
-L FILE	True if file is a symbolic link.
-k FILE	True if file has its 'sticky' bit set.
-p FILE	True if file is a named pipe.
-r FILE	True if file is readable by you.
-s FILE	True if file exists and is not empty.
-S FILE	True if file is a socket.
-t FD	True if FD is opened on a terminal.
-u FILE	True if the file is set-user-id.
-w FILE	True if the file is writable by you.
-x FILE	True if the file is executable by you.
-O FILE	True if the file is effectively owned by you.
-G FILE	True if the file is effectively owned by your group.
-N FILE	True if the file has been modified since it was last read.

FILE1 -nt FILE2 True if file1 is newer than file2 (according to modification date).

FILE1 -ot FILE2 True if file1 is older than file2.

Вот тут например `-z` проверка на нулевую длину аргумента. Далее выходим `exit` с кодом 6. Код 0 успешно код 1 неуспешно.

```
if [ -z $1 ]
then
  echo 'there is empty argument'
  exit 6
fi
echo 'hello'
```

Можно указать ветку else

```
if [ -z $1 ]
then
  echo 'there is empty argument'
  exit 6
else
```

```
echo 'hello'
fi
```

Тут обратите внимание на конструкцию `elif` внутри `if` секции.

```
if [ $1 = yes ]
then
    echo ok
elif [ $1 = no ]
then
    echo not ok
else
    echo sorry
fi
```

можно писать в одну строчку

```
if [ -f /etc/hosts ]; then echo file exists; fi
```

внутри `if` может быть любая команда. Происходит анализ кода завершения этой команды, если этот код 0, тогда выполняется тело `if`. Выше мы использовали команду `test` в форме `[]`. Можно писать просто `test`. А также что-то такое.

```
if grep student /etc/passwd
then
    echo some text
fi
```

можно использовать операторы И `&&` или ИЛИ `||`

```
if [ -d $1 ] && [ -f $2 ]
then
    echo $1
fi
```

можно использовать эти операторы для того чтобы сокращать вид ифа. Т.к тест проверяет возвращаемые код, в случае если первая команда выполнится с кодом 0, будет выполнена вторая

```
[ -f /etc/hosts ] && echo file exists
```

Есть еще один способ писать выражения, через `[[]]`. При этом будет использоваться не `test`, а еще один механизм встроенный в баш. Примеры:

```
[[ $VAR1 = yes && $VAR2 = red ]]
[[ 1 < 2 ]]
[[ -e $b ]]

[[ $var = img* && ($var = *.png || $var = *.jpg) ]] && echo $var starts
with img and ends with .jpg or .png
```

case

Если ответ на вопрос `yes` или `oui` пишем `nice`, если ответ `no` пишем `no`, иначе пишем `okay`.

```
#!/bin/bash

echo are you good?
read GOOD
GOOD=$(echo $GOOD | tr [:upper:] [:lower:])

case $GOOD in
yes|oui)
    echo nice
    ;;
no)
    echo not nice
    ;;
*)
    echo okay
    ;;
esac
```

Циклы

for

Итерируемся по массиву, либо по списку файлов.

однострочный цикл бежит по всем аргументам скрипта

```
for i in $@; do echo $i; done
```

сделаем хитрый массив `ganje` и проитерируемся по нему

```
for i in {115..127}; do echo $i; done
```

сделаем на основе цикла пинг

```
for i in {115..127}; do ping -c 1 192.168.0.$i; done
```

```
for i in anna liza andrey
do
    grep $i /etc/passwd > /dev/null 2>&1 || echo $i user does not exist
done
```

```
#!/bin/bash

if [ -z $1 ]
then
    echo you have to provide at least one argument
    exit 3
fi

MEMFREE=$(free -m | grep Mem | awk '{ print $4 }')

if [ $MEMFREE -lt 256 ]
then
    echo insufficient memory available
    exit 4
fi

sudo apt install -y "$@"

for s in "$@"
do
    sudo systemctl enable --now $s
done
```

while

```
while true; do true; done
```

```
while true
do
```

```
echo true
done
```

```
#!/bin/bash

COUNTER=$1
COUNTER=$(( COUNTER * 60))

minusone(){
    COUNTER=$(( COUNTER - 1))
    sleep 1
}

while [ $COUNTER -gt 0 ]
do
    echo you still have $COUNTER seconds left
    minusone
done

[ $COUNTER = 0 ] && echo time is up && minusone
[ $COUNTER = "-1" ] && echo you now are one second late && minusone

while true
do
    echo you now are ${COUNTER#-} second late
    minusone
done
```

until (opposite while)

```
until who | grep $1; do echo $1 in not logged in; done
```

```
until who | grep $1
do
    echo $1 in not logged in
done
```

break and continue

```
#!/bin/bash

# backup script that stops if insufficient disk space is available

if [ -z $1 ]
```

```

then
    echo enter the name of a directory to back up
    read dir
else
    dir=$1
fi

[ -d ${dir}.backup ] || mkdir ${dir}.backup

for file in $dir/*
do
    used=$( df $dir | tail -1 | awk '{ print $5 }' | sed 's/%//' )
    if [ $used -gt 98 ]
    then
        echo stopping: low disk space
        break
    fi

    cp $file ${dir}.backup
done

```

```

#!/bin/bash

# convert file names to lower case if required

FILES=$(ls)

for file in $FILES
do
    if [[ "$file" != *[:upper:]* ]]; then
        echo "$file" doesn't contain uppercase
        continue
    fi

    OLD="$file"
    NEW=$(echo $file | tr '[:upper:]' '[:lower:]')

    mv "$OLD" "$NEW"
    echo "$OLD has been renamed to $NEW"
done

```

Menu

```

#!/bin/bash

PS3='Enter your choice: '
options=("Option 1" "Option 2" "Option 3" "Quit")

```

```

select opt in "${options[@]}"
do
    case $opt in
        "Option 1")
            echo "you have selected option 1"
            ;;
        "Option 2")
            echo "you have selected option 2"
            ;;
        "Option 3")
            echo "you have selected $REPLY with is $opt"
            ;;
        "Quit")
            break
            ;;
        *) echo "invalid option $REPLY";;
    esac
done

```

```

student@student-virtual-machine:~/bash-scripting$ ./mymenu
1) Option 1
2) Option 2
3) Option 3
4) Quit
Enter your choice: 3
you have selected 3 with is Option 3
Enter your choice: 2
you have selected option 2
Enter your choice: 6
invalid option 6
Enter your choice: ^C

```

- Control C to interrupt.

trap

Use to catch information about signals (except -9).

For help `man 7 signal` and `trap -l`

```

#!/bin/bash
trap "echo ignoring signal" SIGINT SIGTERM
echo pid is $$

while :
do
    sleep 60
done

```

`EXIT` for normal exit.

```
#!/bin/bash
tempfile=/tmp/tmpdata
touch $tempfile
ls -l $tempfile
trap "rm -f $tempfile" EXIT
```

Arrays

Let's compare. We want to copy list of files to directory. With simple string it will not work with names with spaces.

```
files=$(ls *.doc); cp $files ~/backup
```

but with array it will work

```
files=(*.txt); cp "${files[@]}" ~/backup
```

Types of arrays:

- indexed array.
 - do not required using `declare`
 - `my_array=(one two three)`
 - don't confuse with command substitution. `myname=$(whoami)`
- associative array.
 - `declare -A`
 - key value
 - reference to one value of such array `${value[XYZ]}`
 - ordering is not guaranteed

Reference to array values

always use quotes.

- Reference to All elements in the array.

```
"${myarray[@]}"
```

- Reference to second element

```
"${myarray[1]}"
```


- Reference to all keys in array

```
"${!myarray[@]}"
```

Example script indexed array

```
#!/bin/bash
my_array=( a b c )

# print index value 1
echo ${my_array[1]}

# print all items in the array
echo ${my_array[@]}
echo ${my_array[*]}

# print all index values and not their value
echo ${!my_array[@]}

# print the length of the array
echo ${#my_array[@]}

# loop over all items in the array; printing all keys as well as all values
for i in "${!my_array[@]}"
do
    echo "$i" "${my_array[$i]}"
done

# loop on just the values and not the keys
for i in "${my_array[@]}"
do
    echo "$i"
done

# adding a value at a specific position
# using 9 to make sure it is last
my_array[9]=d
echo ${my_array[@]}
echo ${my_array[9]}

# adding items to the end of the array, using the first available index
my_array+=( e f )
for i in "${!my_array[@]}"
do
    echo "$i" "${my_array[$i]}"
done
```

```

student@student-virtual-machine:~/bash-scripting$ ./array1
b
a b c
a b c
0 1 2
3
0 a
1 b
2 c
a
b
c
a b c d
d
0 a
1 b
2 c
9 d
10 e
11 f

```

Example of associative array

```

declare -A my_array

my_array=([value1]=cow [value2]=sheep)

```

Reading output to an array

- `mapfile`

```

mapfile -t my_array <<(my_command)

```

- `cycle with IFS and read`

```

my_array=()
while IFS= read -r line; do
    my_array+=( "$line" )
done < <(my_command)

```

- `readarray`

```
readarray -t my_array < <(seq 5)
declare -p my_array # print array
# output: declare -a my_array=([0]="1" [1]="2" [2]="3" [3]="4" [4]="5" )
```

```
readarray -t myfiles < <(ls)
declare -p myfiles # print array
```

with `cycle`

```
#!/bin/bash

# scanning hosts on $NETWORK
echo enter the IP address of the network that you want to scan for
available hosts
read NETWORK

# enabling some debugging so that we see what happens
set -x
hosts=()
# below IFS is set at the same line as the read statement to make sure it
affects the read statement only
# IFS is set to a space to make sure that as long as it finds a space after
an item the script continues
while IFS= read -r line; do
    hosts+=( "$line" )
done < <( nmap -sn ${NETWORK}/24 | grep ${NETWORK%.*} | awk '{ print $5 }')
set +x

# the two lines below are for debugging only
echo press enter to continue
read

# and here we check that the array works as intended
for value in "${hosts[@]}"
do
    echo $value
```

with `mapfile`

```
#!/bin/bash

# generating SSH key for local user
[ -f /etc/.ssh/id_rsa ] || ssh-keygen

# scanning hosts on $NETWORK
echo enter the IP address of the network that you want to scan for
```

```

available hosts
read NETWORK

# you can fill an array with command output in two ways. The lines below
# are not as efficient but also work
#hosts=()
#while IFS= read -r line; do
#  hosts+=( "$line" )
#done < <( nmap -sn ${NETWORK}/24 | grep ${NETWORK%.*} | awk '{ print $5
#}')

# alternative notation
mapfile -t hosts < <(nmap -sn ${NETWORK}/24 | grep ${NETWORK%.*} | awk '{
print $5 }')

# this line shows debug information; useful while developing but can be
# removed now
for value in "${hosts[@]}"
do
    echo $value
done

PS3='which host do you want to setup? (Ctrl-C to quit) '
select host in "${hosts[@]}"
do
    case $host in
        *)
            echo you selected $host
            set -v
            ssh-copy-id root@$host
            scp /etc/hosts root@$host:/etc
            set +v
            echo this is enough for the proof of concept script
            ;;
    esac
done

```

Looping on array

```

#!/bin/bash
# poem.sh: Pretty-prints one of the ABS Guide author's favorite poems.
# credits: TLDP Advanced Bash Scripting Guide

# Lines of the poem (single stanza).
Line[1]="I do not know which to prefer,"
Line[2]="The beauty of inflections"
Line[3]="Or the beauty of innuendoes,"
Line[4]="The blackbird whistling"
Line[5]="Or just after."
# Note that quoting permits embedding whitespace.

```

```

# Attribution.
Attrib[1]=" Wallace Stevens"
Attrib[2]="\"Thirteen Ways of Looking at a Blackbird\""
# This poem is in the Public Domain (copyright expired).

echo

tput bold    # Bold print.

for index in 1 2 3 4 5    # Five lines.
do
    printf "    %s\n" "${Line[index]}"
done

for index in 1 2          # Two attribution lines.
do
    printf "    %s\n" "${Attrib[index]}"
done

tput sgr0    # Reset terminal.
             # See 'tput' docs.

echo

exit 0

```

Lab about arrays

```

#!/bin/bash

# use readarray to create the associative names
echo enter names for Janitors from Mon-Sun \ (seven names required\ )
read name1 name2 name3 name4 name5 name6 name7

declare -A roster
roster[monday]=$name1
roster[tuesday]=$name2
roster[wednesday]=$name3
roster[thursday]=$name4
roster[friday]=$name5
roster[saturday]=$name6
roster[sunday]=$name7

# print the names of responsible janitors for each day
for i in "${!roster[@]}"
do
    echo "$i" "${roster[$i]}"
done

```

with order array

```
#!/bin/bash

# use readarray to create the associative names
echo enter names for Janitors from Mon-Sun \ (seven names required\ )
read name1 name2 name3 name4 name5 name6 name7

declare -A roster; declare -a order
roster[monday]=$name1; order+=( "monday" )
roster[tuesday]=$name2; order+=( "tuesday" )
roster[wednesday]=$name3; order+=( "wednesday" )
roster[thursday]=$name4; order+=( "thursday" )
roster[friday]=$name5; order+=( "friday" )
roster[saturday]=$name6; order+=( "saturday" )
roster[sunday]=$name7; order+=( "sunday" )

# print the names of responsible janitors for each day
for i in "${order[@]}"
do
    echo "$i" "${roster[$i]}"
done
```

Best Practices

- Starting with comments line
- Write and test each part of the script separately
- Use break lines
- use `set -x` and `set +x` to debug section. It's useful to debug all parts of scripts.
- use combination `echo just added the user press enter` and `read` to a problematic areas

Using BASH_XTRACEFD for debug

```
#!/bin/bash

# Use FD 15 to capture the debug stream caused by "set -x". No magic in 15
# number, it's just more than 0 1 and 2 like stdin and etc.
exec 15>/tmp/bash-debug.log
# Tell bash about it (there's nothing special about 15, it's arbitrary)
export BASH_XTRACEFD=15

# turn on debugging:
set -x

# run some commands:
cd /etc
find
echo "that was it"
```

```
# Close the debugging:
set +x

# Close the file descriptor
exec 15>&-

# See what we got:
cat /tmp/bash-debug.log
```

Complex scripts

trivial restart if service down

```
#!/bin/bash
#
# Monitoring process
#
COUNTER=0
while ps aux | grep $1 | grep -v grep | grep -v $0 > /dev/null
do
    COUNTER=$((COUNTER+1))
    sleep 1
    echo COUNTER is $COUNTER
done

logger HTTPMONITOR: $1 stopped at `date`
systemctl start $1
```

check users are created

```
#!/bin/bash

# RHCSA labs grading script - SvV
# version 0.1

# verify password settings
grep 'PASS_MIN_LEN      6' /etc/login.defs >/dev/null 2>&1 || echo you did
not set minimal password length to 6
grep 'PASS_MAX_DAYS     90' /etc/login.defs >/dev/null 2>&1 || echo max
password validity is not set to 90 days

# verify new users
for i in anna audrey linda lisa
do
    grep $i /etc/passwd >/dev/null 2>&1 || echo user $i does not exist
done
```

```

#verify new file in user homedirs
for i in anna audrey linda lisa
do
    ls /home/$i/newfile >/dev/null 2>&1 || echo no newfile in $i home
directory
done

# verify user group membership
id anna | grep profs >/dev/null 2>&1 || echo anna is not a member of group
profs
id audrey | grep profs >/dev/null 2>&1 || echo audrey is not a member of
group profs
id linda | grep sales >/dev/null 2>&1 || echo linda is not a member of
group sales
id lisa | grep sales >/dev/null 2>&1 || echo lisa is not a member of group
sales

# check that accounts linda and lisa are locked
passwd -S linda | grep locked >/dev/null 2>&1|| echo user linda password is
not locked
passwd -S lisa | grep locked>/dev/null 2>&1 || echo user linda password is
not locked

# evaluate passwords for anna and audrey
sshpass -p "password" ssh -o StrictHostKeyChecking=no anna@localhost exit
>/dev/null 2>&1 || echo password for anna not set correctly
sshpass -p "password" ssh -o StrictHostKeyChecking=no audrey@localhost exit
>/dev/null 2>&1 || echo password for audrey not set correctly

echo grading completed

```

install kube cluster

```

#!/bin/bash
#
# verified on Fedora 31, 33 and Ubuntu LTS 20.04

echo this script works on Fedora 31, 33 and Ubuntu 20.04
echo it does NOT currently work on Fedora 32
echo it requires the machine where you run it to have 6GB of RAM or more
echo press Enter to continue
read

#####
echo #####
echo WARNING
echo #####
echo Nov 2020 - currently this script is NOT supported on Mac OS Big Sur
echo I will communicate here one Apple/VMware have provided updates that

```



```

make it work again
echo
echo Check the Setup Guide provided in this repository for alternative
installations
echo
echo press Enter to continue
read

# setting MYOS variable
MYOS=$(hostnamectl | awk '/Operating/ { print $3 }')
OSVERSION=$(hostnamectl | awk '/Operating/ { print $4 }')

egrep '^flags.*(vmx|svm)' /proc/cpuinfo || (echo enable CPU virtualization
support and try again && exit 9)

# debug MYOS variable
echo MYOS is set to $MYOS

#### Fedora config
if [ $MYOS = "Fedora" ]
then
    if [ $OSVERSION = 32 ]
    then
        echo Fedora 32 is not currently supported
        exit 9
    fi

    sudo dnf clean all
    sudo dnf -y upgrade

    # install KVM software
    sudo dnf install @virtualization -y
    sudo systemctl enable --now libvirtd
    sudo usermod -aG libvirt `id -un`
fi

### Ubuntu config
if [ $MYOS = "Ubuntu" ]
then
    sudo apt-get update -y
    sudo apt-get install -y apt-transport-https curl
    sudo apt-get upgrade -y
    sudo apt-get install -y qemu-kvm libvirt-daemon-system libvirt-clients
bridge-utils

    sudo adduser `id -un` libvirt
    sudo adduser `id -un` kvm
fi

# install kubect1
curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s
https://storage.googleapis.com/kubernetes-
release/release/stable.txt`/bin/linux/amd64/kubect1
chmod +x ./kubect1

```

```
sudo mv ./kubectl /usr/local/bin/kubectl

# install minikube
echo downloading minikube, check version
curl -Lo minikube
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-
amd64

sudo chmod +x minikube
sudo mv minikube /usr/local/bin

# start minikube
minikube start --memory 4096 --vm-driver=kvm2

echo if this script ends with an error, restart the virtual machine
echo and manually run minikube start --memory 4096 --vm-driver=kvm2
```