3.2. Variables

3.2.1. Types of variables

As seen in the examples above, shell variables are in uppercase characters by convention. Bash keeps a list of two types of variables:

3.2.1.1. Global variables

Global variables or environment variables are available in all shells. The **env** or **printenv** commands can be used to display environment variables. These programs come with the *sh-utils* package.

Below is a typical output:

```
franky ~> printenv
CC=gcc
CDPATH=.:~:/usr/local:/usr:/
CFLAGS=-O2 -fomit-frame-pointer
COLORTERM=gnome-terminal
CXXFLAGS=-O2 -fomit-frame-pointer
DISPLAY=:0
DOMAIN=hq.garrels.be
e=
TOR=vi
FCEDIT=vi
FIGNORE=.o:~
G_BROKEN_FILENAMES=1
GDK_USE_XFT=1
GDMSESSION=Default
GNOME_DESKTOP_SESSION_ID=Default
GTK_RC_FILES=/etc/gtk/gtkrc:/nethome/franky/.gtkrc-1.2-gnome2
GWMCOLOR=darkgreen
GWMTERM=xterm
HISTFILESIZE=5000
history_control=ignoredups
HISTSIZE=2000
HOME=/nethome/franky
HOSTNAME=octarine.hq.garrels.be
INPUTRC=/etc/inputrc
IRCNAME=franky
JAVA_HOME=/usr/java/j2sdk1.4.0
LANG=en_US
LDFLAGS=-s
LD_LIBRARY_PATH=/usr/lib/mozilla:/usr/lib/mozilla/plugins
LESSCHARSET=latin1
LESS=-edfMQ
LESSOPEN=|/usr/bin/lesspipe.sh %s
LEX=flex
LOCAL_MACHINE=octarine
LOGNAME=franky
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01:or=01;05;37;41:mi=01;05;37;41:ex=01;32:*.cmd=01;32:*.exe=01
MACHINES=octarine
MAILCHECK=60
MAIL=/var/mail/franky
MANPATH=/usr/man:/usr/share/man/:/usr/local/man:/usr/X11R6/man
MEAN_MACHINES=octarine
MOZ_DIST_BIN=/usr/lib/mozilla
MOZILLA_FIVE_HOME=/usr/lib/mozilla
MOZ_PROGRAM=/usr/lib/mozilla/mozilla-bin
MTOOLS_FAT_COMPATIBILITY=1
MYMALLOC=0
NNTPPORT=119
NNTPSERVER=news
NPX_PLUGIN_PATH=/plugin/ns4plugin/:/usr/lib/netscape/plugins
OLDPWD=/nethome/franky
OS=Linux
PAGER=less
PATH=/nethome/franky/bin.Linux:/nethome/franky/bin:/usr/local/bin:/usr/local/sbin:/usr/X11R6/bin:/usr/bin:/usr/sbin:/bin:/sbin:.
PS1=\[\033[1;44m\]franky is in \w\[\033[0m\]
PS2=More input>
PWD=/nethome/franky
SESSION_MANAGER=local/octarine.hq.garrels.be:/tmp/.ICE-unix/22106
SHELL=/bin/bash
SHELL_LOGIN=--login
SHLVL=2
SSH_AGENT_PID=22161
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SSH_AUTH_SOCK=/tmp/ssh-XXmhQ4fC/agent.22106
START_WM=twm
TERM=xterm
TYPE=type
USERNAME=franky
USER=franky
_=/usr/bin/printenv
VISUAL=vi
WINDOWID=20971661
XAPPLRESDIR=/nethome/franky/app-defaults
XAUTHORITY=/nethome/franky/.Xauthority
```

```
XENVIRONMENT=/nethome/franky/.Xdefaults
XFILESEARCHPATH=/usr/X11R6/lib/X11/%L/%T/%N%C%S:/usr/X11R6/lib/X11/%l/%T/%N%C%S:/usr/X11R6/lib/X11/%T/%N%C%S:/usr/X11R6/lib/X11/%L/%T/%N%S:/u
XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB
XMODIFIERS=@im=none
XTERMID=
XWINHOME=/usr/X11R6
X=X11R6
YACC=bison -y
```

3.2.1.2. Local variables

Local variables are only available in the current shell. Using the **set** built-in command without any options will display a list of all variables (including environment variables) and functions. The output will be sorted according to the current locale and displayed in a reusable format.

Below is a diff file made by comparing **printenv** and **set** output, after leaving out the functions which are also displayed by the **set** command:

```
franky ~> diff set.sorted printenv.sorted | grep "<" | awk '{ print $2 }'
BASE=/nethome/franky/.Shell/hq.garrels.be/octarine.aliases
BASH=/bin/bash
BASH_VERSINFO=([0]="2"
BASH_VERSION='2.05b.0(1)-release'
COLUMNS=80
DIRSTACK=()
DO_FORTUNE=
EUID=504
GROUPS=()
HERE=/home/franky
HISTFILE=/nethome/franky/.bash_history
HOSTTYPE=i686
IFS=$'
LINES=24
MACHTYPE=i686-pc-linux-gnu
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
PIPESTATUS=([0]="0")
PPID=10099
PS4='+
PWD_REAL='pwd
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
THERE=/home/franky
UID=504
```

Awk

the GNU Awk programming language is explained in Chapter 6.

3.2.1.3. Variables by content

Apart from dividing variables in local and global variables, we can also divide them in categories according to the sort of content the variable contains. In this respect, variables come in 4 types:

- String variables
- Integer variables
- Constant variables
- Array variables

We'll discuss these types in Chapter 10. For now, we will work with integer and string values for our variables.

3.2.2. Creating variables

Variables are case sensitive and capitalized by default. Giving local variables a lowercase name is a convention which is sometimes applied. However, you are free to use the names you want or to mix cases. Variables can also contain digits, but a name starting with a digit is not allowed:

```
prompt> export 1number=1
bash: export: `1number=1': not a valid identifier
```

To set a variable in the shell, use

VARNAME="value"

Putting spaces around the equal sign will cause errors. It is a good habit to quote content strings when assigning values to variables: this will reduce the chance that you make errors.

Some examples using upper and lower cases, numbers and spaces:

```
franky ~> MYVAR1="2"

franky ~> echo $MYVAR1
2

franky ~> first_name="Franky"

franky ~> echo $first_name
Franky
```

```
franky ~> full_name="Franky M. Singh"

franky ~> echo $full_name
Franky M. Singh

franky ~> MYVAR-2="2"
bash: MYVAR-2=2: command not found

franky ~> MYVAR1 ="2"
bash: MYVAR1: command not found

franky ~> MYVAR1= "2"
bash: 2: command not found

franky ~> unset MYVAR1 first_name full_name

franky ~> echo $MYVAR1 $first_name $full_name
<--no output-->

franky ~>
```

3.2.3. Exporting variables

A variable created like the ones in the example above is only available to the current shell. It is a local variable: child processes of the current shell will not be aware of this variable. In order to pass variables to a subshell, we need to *export* them using the **export** built-in command. Variables that are exported are referred to as environment variables. Setting and exporting is usually done in one step:

export VARNAME="*value*"

A subshell can change variables it inherited from the parent, but the changes made by the child don't affect the parent. This is demonstrated in the example:

```
franky ~> full_name="Franky M. Singh"

franky ~> bash

franky ~> echo $full_name


franky ~> exit

franky ~> export full_name

franky ~> bash

franky ~> echo $full_name
Franky M. Singh

franky ~> export full_name="Charles the Great"

franky ~> echo $full_name
Charles the Great

franky ~> exit

franky ~> echo $full_name
Franky M. Singh

franky ~>
```

When first trying to read the value of `full_name` in a subshell, it is not there (**echo** shows a null string). The subshell quits, and `full_name` is exported in the parent - a variable can be exported after it has been assigned a value. Then a new subshell is started, in which the variable exported from the parent is visible. The variable is changed to hold another name, but the value for this variable in the parent stays the same.

3.2.4. Reserved variables

3.2.4.1. Bourne shell reserved variables

Bash uses certain shell variables in the same way as the Bourne shell. In some cases, Bash assigns a default value to the variable. The table below gives an overview of these plain shell variables:

Table 3-1. Reserved Bourne shell variables

Variable name	Definition
CDPATH	A colon-separated list of directories used as a search path for the **cd** built-in command.
HOME	The current user's home directory; the default for the **cd** built-in. The value of this variable is also used by tilde expansion.
IFS	A list of characters that separate fields; used when the shell splits words as part of expansion.
MAIL	If this parameter is set to a file name and the `MAILPATH` variable is not set, Bash informs the user of the arrival of mail in the specified file.
MAILPATH	A colon-separated list of file names which the shell periodically checks for new mail.
OPTARG	The value of the last option argument processed by the **getopts** built-in.
OPTIND	The index of the last option argument processed by the **getopts** built-in.
PATH	A colon-separated list of directories in which the shell looks for commands.
PS1	The primary prompt string. The default value is "'\s-\v\$ '".
PS2	The secondary prompt string. The default value is "'> '".

3.2.4.2. Bash reserved variables

These variables are set or used by Bash, but other shells do not normally treat them specially.

Table 3-2. Reserved Bash variables

Variable name	Definition
auto_resume	This variable controls how the shell interacts with the user and job control.
BASH	The full pathname used to execute the current instance of Bash.
BASH_ENV	If this variable is set when Bash is invoked to execute a shell script, its value is expanded and used as the name of a startup file to read before executing the script.
BASH_VERSION	The version number of the current instance of Bash.
BASH_VERSINFO	A read-only array variable whose members hold version information for this instance of Bash.
COLUMNS	Used by the **select** built-in to determine the terminal width when printing selection lists. Automatically set upon receipt of a *SIGWINCH* signal.
COMP_CWORD	An index into `${COMP_WORDS}` of the word containing the current cursor position.
COMP_LINE	The current command line.
COMP_POINT	The index of the current cursor position relative to the beginning of the current command.
COMP_WORDS	An array variable consisting of the individual words in the current command line.
COMPREPLY	An array variable from which Bash reads the possible completions generated by a shell function invoked by the programmable completion facility.
DIRSTACK	An array variable containing the current contents of the directory stack.
EUID	The numeric effective user ID of the current user.
FCEDIT	The editor used as a default by the `-e` option to the **fc** built-in command.
FIGNORE	A colon-separated list of suffixes to ignore when performing file name completion.
FUNCNAME	The name of any currently-executing shell function.
GLOBIGNORE	A colon-separated list of patterns defining the set of file names to be ignored by file name expansion.
GROUPS	An array variable containing the list of groups of which the current user is a member.
histchars	Up to three characters which control history expansion, quick substitution, and *tokenization*.
HISTCMD	The history number, or index in the history list, of the current command.
HISTCONTROL	Defines whether a command is added to the history file.
HISTFILE	The name of the file to which the command history is saved. The default value is `~/.bash_history`.
HISTFILESIZE	The maximum number of lines contained in the history file, defaults to 500.
HISTIGNORE	A colon-separated list of patterns used to decide which command lines should be saved in the history list.
HISTSIZE	The maximum number of commands to remember on the history list, default is 500.
HOSTFILE	Contains the name of a file in the same format as `/etc/hosts` that should be read when the shell needs to complete a hostname.
HOSTNAME	The name of the current host.
HOSTTYPE	A string describing the machine Bash is running on.
IGNOREEOF	Controls the action of the shell on receipt of an *EOF* character as the sole input.
INPUTRC	The name of the Readline initialization file, overriding the default `/etc/inputrc`.
LANG	Used to determine the locale category for any category not specifically selected with a variable starting with `LC_`.
LC_ALL	This variable overrides the value of `LANG` and any other `LC_` variable specifying a locale category.
LC_COLLATE	This variable determines the collation order used when sorting the results of file name expansion, and determines the behavior of range expressions, equivalence classes, and collating sequences within file name expansion and pattern matching.
LC_CTYPE	This variable determines the interpretation of characters and the behavior of character classes within file name expansion and pattern matching.
LC_MESSAGES	This variable determines the locale used to translate double-quoted strings preceded by a "$" sign.
LC_NUMERIC	This variable determines the locale category used for number formatting.
LINENO	The line number in the script or shell function currently executing.
LINES	Used by the **select** built-in to determine the column length for printing selection lists.
MACHTYPE	A string that fully describes the system type on which Bash is executing, in the standard GNU CPU-COMPANY-SYSTEM format.
MAILCHECK	How often (in seconds) that the shell should check for mail in the files specified in the `MAILPATH` or `MAIL` variables.
OLDPWD	The previous working directory as set by the **cd** built-in.
OPTERR	If set to the value 1, Bash displays error messages generated by the **getopts** built-in.
OSTYPE	A string describing the operating system Bash is running on.
PIPESTATUS	An array variable containing a list of exit status values from the processes in the most recently executed foreground pipeline (which may contain only a single command).
POSIXLY_CORRECT	If this variable is in the environment when **bash** starts, the shell enters POSIX mode.
PPID	The process ID of the shell's parent process.
PROMPT_COMMAND	If set, the value is interpreted as a command to execute before the printing of each primary prompt (`PS1`).
PS3	The value of this variable is used as the prompt for the **select** command. Defaults to "'#? '"
PS4	The value is the prompt printed before the command line is echoed when the `-x` option is set; defaults to "'+ '".
PWD	The current working directory as set by the **cd** built-in command.
	Each time this parameter is referenced, a random integer between 0 and 32767 is generated. Assigning a value to this variable seeds the random

RANDOM	number generator.
REPLY	The default variable for the **read** built-in.
SECONDS	This variable expands to the number of seconds since the shell was started.
SHELLOPTS	A colon-separated list of enabled shell options.
SHLVL	Incremented by one each time a new instance of Bash is started.
TIMEFORMAT	The value of this parameter is used as a format string specifying how the timing information for pipelines prefixed with the **time** reserved word should be displayed.
TMOUT	If set to a value greater than zero, TMOUT is treated as the default timeout for the **read** built-in. In an interative shell, the value is interpreted as the number of seconds to wait for input after issuing the primary prompt when the shell is interactive. Bash terminates after that number of seconds if input does not arrive.
UID	The numeric, real user ID of the current user.

Check the Bash man, info or doc pages for extended information. Some variables are read-only, some are set automatically and some lose their meaning when set to a different value than the default.

3.2.5. Special parameters

The shell treats several parameters specially. These parameters may only be referenced; assignment to them is not allowed.

Table 3-3. Special bash variables

Character	Definition
$*	Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the IFS special variable.
$@	Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands to a separate word.
$#	Expands to the number of positional parameters in decimal.
$?	Expands to the exit status of the most recently executed foreground pipeline.
$-	A hyphen expands to the current option flags as specified upon invocation, by the **set** built-in command, or those set by the shell itself (such as the -i).
$$	Expands to the process ID of the shell.
$!	Expands to the process ID of the most recently executed background (asynchronous) command.
$0	Expands to the name of the shell or shell script.
$_	The underscore variable is set at shell startup and contains the absolute file name of the shell or script being executed as passed in the argument list. Subsequently, it expands to the last argument to the previous command, after expansion. It is also set to the full pathname of each command executed and placed in the environment exported to that command. When checking mail, this parameter holds the name of the mail file.

$* vs. $@

The implementation of "$*" has always been a problem and realistically should have been replaced with the behavior of "$@". In almost every case where coders use "$*", they mean "$@". "$*" Can cause bugs and even security holes in your software.

The positional parameters are the words following the name of a shell script. They are put into the variables $1, $2, $3 and so on. As long as needed, variables are added to an internal array. $# holds the total number of parameters, as is demonstrated with this simple script:

```
#!/bin/bash

# positional.sh
# This script reads 3 positional parameters and prints them out.

POSPAR1="$1"
POSPAR2="$2"
POSPAR3="$3"

echo "$1 is the first positional parameter, \$1."
echo "$2 is the second positional parameter, \$2."
echo "$3 is the third positional parameter, \$3."
echo
echo "The total number of positional parameters is $#."
```

Upon execution one could give any numbers of arguments:

```
franky ~> positional.sh one two three four five
one is the first positional parameter, $1.
two is the second positional parameter, $2.
three is the third positional parameter, $3.

The total number of positional parameters is 5.

franky ~> positional.sh one two
one is the first positional parameter, $1.
two is the second positional parameter, $2.
 is the third positional parameter, $3.

The total number of positional parameters is 2.
```

More on evaluating these parameters is in Chapter 7 and Section 9.7.

Some examples on the other special parameters:

```
franky ~> grep dictionary /usr/share/dict/words
```

```
dictionary

franky ~> echo $_
/usr/share/dict/words

franky ~> echo $$
10662

franky ~> mozilla &
[1] 11064

franky ~> echo $!
11064

franky ~> echo $0
bash

franky ~> echo $?
0

franky ~> ls doesnotexist
ls: doesnotexist: No such file or directory

franky ~> echo $?
1

franky ~>
```

User *franky* starts entering the **grep** command, which results in the assignment of the _ variable. The process ID of his shell is 10662. After putting a job in the background, the `!` holds the process ID of the backgrounded job. The shell running is **bash**. When a mistake is made, `?` holds an exit code different from 0 (zero).

3.2.6. Script recycling with variables

Apart from making the script more readable, variables will also enable you to faster apply a script in another environment or for another purpose. Consider the following example, a very simple script that makes a backup of *franky*'s home directory to a remote server:

```
#!/bin/bash

# This script makes a backup of my home directory.

cd /home

# This creates the archive
tar cf /var/tmp/home_franky.tar franky > /dev/null 2>&1

# First remove the old bzip2 file.  Redirect errors because this generates some if the archive
# does not exist.  Then create a new compressed file.
rm /var/tmp/home_franky.tar.bz2 2> /dev/null
bzip2 /var/tmp/home_franky.tar

# Copy the file to another host - we have ssh keys for making this work without intervention.
scp /var/tmp/home_franky.tar.bz2 bordeaux:/opt/backup/franky > /dev/null 2>&1

# Create a timestamp in a logfile.
date >> /home/franky/log/home_backup.log
echo backup succeeded >> /home/franky/log/home_backup.log
```

First of all, you are more likely to make errors if you name files and directories manually each time you need them. Secondly, suppose *franky* wants to give this script to *carol*, then carol will have to do quite some editing before she can use the script to back up her home directory. The same is true if *franky* wants to use this script for backing up other directories. For easy recycling, make all files, directories, usernames, servernames etcetera variable. Thus, you only need to edit a value once, without having to go through the entire script to check where a parameter occurs. This is an example:

```
#!/bin/bash

# This script makes a backup of my home directory.

# Change the values of the variables to make the script work for you:
BACKUPDIR=/home
BACKUPFILES=franky
TARFILE=/var/tmp/home_franky.tar
BZIPFILE=/var/tmp/home_franky.tar.bz2
SERVER=bordeaux
REMOTEDIR=/opt/backup/franky
LOGFILE=/home/franky/log/home_backup.log

cd $BACKUPDIR

# This creates the archive
tar cf $TARFILE $BACKUPFILES > /dev/null 2>&1

# First remove the old bzip2 file.  Redirect errors because this generates some if the archive
# does not exist.  Then create a new compressed file.
rm $BZIPFILE 2> /dev/null
bzip2 $TARFILE

# Copy the file to another host - we have ssh keys for making this work without intervention.
scp $BZIPFILE $SERVER:$REMOTEDIR > /dev/null 2>&1

# Create a timestamp in a logfile.
date >> $LOGFILE
echo backup succeeded >> $LOGFILE
```

Large directories and low bandwidth

The above is purely an example that everybody can understand, using a small directory and a host on the same subnet. Depending on your bandwidth, the size of the directory and the location of the remote server, it can take an awful lot of time to make backups using this mechanism. For larger directories and lower bandwidth, use **rsync** to keep the directories at both ends synchronized.

Prev
Shell initialization files

Home
Up

Next
Quoting characters