User Input!

Let's make our scripts interactive.

Introduction

We looked at one form of user input (command line arguments (bash-variables.php#arguments)) in the previous section. Now we would like to introduce other ways the user may provide input to the Bash script. Following this we'll have a discussion on when and where is best to use each method.

After the mammoth previous section this one is much easier to get through.

Ask the User for Input

If we would like to ask the user for input then we use a command called **read**.
This command takes the input and will save it into a variable.

```
read var1
```

Let's look at a simple example:

introduction.sh

```
1.  #!/bin/bash
2.  # Ask the user for their name
3.
4.  echo Hello, who am I talking to?
5.
6.  read varname
7.
8.  echo It\'s nice to meet you $varname
```

Let's break it down:

- **Line 4** - Print a message asking the user for input.
- **Line 6** - Run the command **read** and save the users response into the variable **varname**
- **Line 8** - **echo** another message just to verify the read command worked. Note: I had to put a backslash (\ ) in front of the ' so that it was escaped.

Terminal

```
1.  user@bash: ./introduction.sh
2.  Hello, who am I talking to?
3.  Ryan
4.  It's nice to meet you Ryan
5.  user@bash:
```

- Note: Ryan above is in italics just to show that it was something I typed in. On your terminal input will show up normally.

More with Read

You are able to alter the behaviour of **read** with a variety of command line options. (See the man page for read to see all of them.) Two commonly used options however are **-p** which allows you to specify a prompt and **-s** which makes the input silent. This can make it easy to ask for a username and password combination like the example below:

login.sh

```
1.  #!/bin/bash
2.  # Ask the user for login details
3.
4.  read -p 'Username: ' uservar
5.  read -sp 'Password: ' passvar
6.  echo
7.  echo Thankyou $uservar we now have your login details
```

- On lines 4 and 5 above we include the prompt within quotes so we can have a space included with it. Otherwise the user input will start straight after the last character of the prompt which isn't ideal from a readability point of view.

Terminal

```
1.  user@bash: ./login.sh
2.  Username: ryan
3.  Password:
4.  Thankyou ryan we now have your login details
5.  user@bash:
```

More variables

So far we have looked at a single word as input. We can do more than that however.

cars.sh

```
1.  #!/bin/bash
2.  # Demonstrate how read actually works
3.
4.  echo What cars do you like?
5.
6.  read car1 car2 car3
7.
8.  echo Your first car was: $car1
9.  echo Your second car was: $car2
10.  echo Your third car was: $car3
```

Terminal

```
user@bash: ./cars.sh
What cars do you like?
Jaguar Maserati Bentley
Your first car was: Jaguar
Your second car was: Maserati
Your third car was: Bentley
user@bash:
user@bash: ./cars.sh
What cars do you like?
Jaguar Maserati Bentley Lotus
Your first car was: Jaguar
Your second car was: Maserati
Your third car was: Bentley Lotus
user@bash:
```

The general mechanism is that you can supply several variable names to **read**. Read will then take your input and split it on whitespace. The first item will then be assigned to the first variable name, the second item to the second variable name and so on. If there are more items than variable names then the remaining items will all be added to the last variable name. If there are less items than variable names then the remaining variable names will be set to blank or null.

Reading from STDIN

It's common in Linux to pipe (/linuxtutorial/piping.php#piping) a series of simple, single purpose commands together to create a larger solution tailored to our exact needs. The ability to do this is one of the real strenghs of Linux. It turns out that we can easily accommodate this mechanism with our scripts also. By doing so we can create scripts that act as filters to modify data in specific ways for us.

Bash accomodates piping and redirection by way of special files. Each process gets it's own set of files (one for STDIN, STDOUT and STDERR respectively) and they are linked when piping or redirection is invoked. Each process gets the following files:

- **STDIN** - /proc/<processID>/fd/0
- **STDOUT** - /proc/<processID>/fd/1
- **STDERR** - /proc/<processID>/fd/2

To make life more convenient the system creates some shortcuts for us:

- **STDIN** - /dev/stdin or /proc/self/fd/0
- **STDOUT** - /dev/stdout or /proc/self/fd/1
- **STDERR** - /dev/stderr or /proc/self/fd/2

fd in the paths above stands for file descriptor.

So if we would like to make our script able to process data that is piped to it all we need to do is read the relevant file. All of the files mentioned above behave like normal files.

summary

```
#!/bin/bash
# A basic summary of my sales report
```

```
4.  echo Here is a summary of the sales data:
5.  echo ==================================
6.  echo
7.
8.  cat /dev/stdin | cut -d' ' -f 2,3 | sort
```

Let's break it down:

- **Lines 4, 5, 6** - Print a title for the output
- **Line 8** - **cat** the file representing STDIN, **cut** setting the delimiter to a space, fields 2 and 3 then sort the output.

Terminal

```
1.  user@bash: cat salesdata.txt
2.  Fred apples 20 August 4
3.  Susy oranges 5 August 7
4.  Mark watermelons 12 August 10
5.  Terry peaches 7 August 15
6.  user@bash:
7.  user@bash: cat salesdata.txt | ./summary
8.  Here is a summary of the sales data:
9.  ==================================
10.
11.  apples 20
12.  oranges 5
13.  peaches 7
14.  watermelons 12
15.  user@bash:
```

So which should I use?

So we now have 3 methods for getting input from the user:

- Command line arguments
- Read input during script execution
- Accept data that has been redirected into the Bash script via STDIN

Which method is best depends on the situation.

You should normally favor command line arguments wherever possible. They are the most convenient for users as the data will be stored in their command history so they can easily return to it. It is also the best approach if your script may be called by other scripts or processes (eg. maybe you want it to run periodically using CRON).

Sometimes the nature of the data is such that it would not be ideal for it to be stored in peoples command histories etc. A good example of this is login credentials (username and password). In these circumstances it is best to **read** the data during script execution.

If all the script is doing is processing data in a certain way then it is probably best to work with STDIN. This way it can easily be added into a pipeline.

Sometimes you may find that a combination is ideal. The user may supply a filename as a command line argument and if not then the script will process what it finds on STDIN (when we look at If statements we'll see how this may be achieved). Or maybe command line arguments define certain behaviour but **read** is also used to ask for more information if required.

Ultimatately you should think about 3 factors when deciding how users will supply data to your Bash script:

- **Ease of use** - which of these methods will make it easiest for users to use my script?
- **Security** - Is there sensitive data which I should handle appropriately?
- **Robustness** - Can I make it so that my scripts operation is intuitive and flexible and also make it harder to make simple mistakes?

Summary

Stuff We Learnt

read varName
Read input from the user and store it in the variable varName.

/dev/stdin
A file you can read to get the STDIN for the Bash script

Important Concepts

Usability
Your choice of input methods will have an impact on how useable your script is.

Activities

Let's dabble with input.

- Create a simple script which will ask the user for a few pieces of information then combine this into a message which is echo'd to the screen.
- Add to the previous script to add in some data coming from command line arguments and maybe some of the other system variables.

Bash Scripting Tutorial

(/bash-scripting-tutorial/)

CSS Tutorial

(/css-tutorial/)

Regular Expressions

(/regular-expressions-tutorial/)

Programming Challenges

(/programming-challenges/)

Problem Solving

(/problem-solving-skills/)

Boolean Algebra Tutorial

(/boolean-algebra-tutorial/)

Basic Design Tutorial

(/graphic-design-tutorial/)

Solve the Cube

(/rubiks-cube-tutorial/)

Software Design and Development

(/software-design-and-development/)

Website Development Challenges

(/website-development-challenges/)

Thoughts

(/thoughts/)

micro:bit Tutorial

(/microbit-tutorial/)

- Create a script which will take data from STDIN and print the 3rd line only.
- Now play about with creating a script which will behave as a filter. Create a script which will rearrange the output of the command **ls -l** in a useful way (eg maybe you only print the filename, size and owner) (Hint: awk (/linuxtutorial/bonus.php#awk) can be useful here).

❮ Variables (./bash-variables.php) Arithmetic ❯ (./bash-arithmetic.php)

Home

(/)

Linux Tutorial

(/linuxtutorial/)

HTML Tutorial

(/html-tutorial/)

Binary Tutorial

(/binary-tutorial/)

Education is the kindling of a flame,
not the filling of a vessel.
- *Socrates*

Contact (/contact.php) | Disclaimer (/disclaimer.php)