

Examples

Read the manual page for bash!

If needed, check out my not-so-brief [Introduction to Manual Pages](#) to learn how to reference these manual pages more efficiently.

I would also recommend the book [Bash Pocket Reference by Arnold Robbins](#), it is a pretty dense read but worth looking over. There are a lot of good examples, and it has actually been a pretty useful reference for me to keep close by. It isn't a book, but rather a collection of examples and concepts that are common in bash scripting.

Redirecting Output

Knowing how to control your output streams in bash can help to make you much more effective at writing commands. For example, say you want to run `clion` on the CWD and fork the process to the background.

```
clion . &
[1] 178345
2021-12-18 16:13:28,384 [ 3869] WARN - I.NotificationGroupManagerImpl - Notification group CodeWithMe is
already registered (group=com.intellij.notification.NotificationGroup@68d6e24d). Plugin descriptor:
PluginDescriptor(name=Code With Me, id=com.jetbrains.codeWithMe, descriptorPath=plugin.xml,
path=~/.local/share/JetBrains/Toolbox/apps/CLion/ch-0/213.5744.254/plugins/cwm-plugin,
version=213.5744.254, package=null, isBundled=true)
2021-12-18 16:13:29,647 [ 5132] WARN - pl.local.NativeFileWatcherImpl - Watcher terminated with exit code
130
```

Woah! We've inherited the output from the process we forked to the background, and we've also lost immediate control of the process so `CTRL+C` won't interrupt the program and stop the output as it normally would. To fix this, run `fg` to bring the process back to the foreground, and then try pressing `CTRL+C` again. The process will terminate and the output will stop.

If we want to fork this process to the background and redirect all of its output so we don't need to see it, we can run the following command. Note that in this case, we are redirecting to `/dev/null` to throw away the output. If we wanted to, we could instead redirect to a file and log the output.

```
clion . 2>/dev/null 1>&2 &
```

Or, a shorter version, which is shorthand for redirecting all output.

```
clion . &>/dev/null &
```

If we want to redirect only standard output

```
clion 1>/dev/null &
```

And if we want to redirect only standard error

```
clion. 2>/dev/null
```

Creating Scripts

Bash scripting is much like interacting with the bash terminal - the similarity can be easily seen in how we would split a bash command to multiple lines...

```
kapak@base:~$ l\
> s -la
total 76
drwxr-xr-x 9 username username 4096 Jul 28 01:24 .
drwxr-xr-x 3 root root 4096 Jul 6 09:49 ..
-rw----- 1 username username 5423 Jul 20 18:10 .bash_history
-rw-r--r-- 1 username username 220 Jul 6 09:49 .bash_logout
-rw-r--r-- 1 username username 3771 Jul 6 09:49 .bashrc
...( Reduced Output ) ...
kapak@base:~$ ls -la
total 76
drwxr-xr-x 9 username username 4096 Jul 28 01:24 .
drwxr-xr-x 3 root root 4096 Jul 6 09:49 ..
-rw----- 1 username username 5423 Jul 20 18:10 .bash_history
-rw-r--r-- 1 username username 220 Jul 6 09:49 .bash_logout
-rw-r--r-- 1 username username 3771 Jul 6 09:49 .bashrc
...( Reduced Output ) ...
```

In a bash script, we would handle splitting `ls -la` across multiple lines much the same. Create the file below, name it test.sh -

```
#!/bin/bash
ls -la
l\
s -la
```

Now make the file executable and run the script, you should see the output of `ls -la` twice, since this script is a simple example of splitting commands across lines.

```
# Make the script executable
sudo chmod a+x test.sh

# Run the script
./test.sh
```

This of course isn't a common use case, but it shows how you can use `\` to effectively escape a newline and continue your command on the next line.

Printf Formatting

This is just one of many commands in bash, but you will use it a lot so getting to know the syntax well will make your life a lot easier.

```
#!/bin/bash

## Author: Shaun Reed | Contact: shaunrd0@gmail.com | URL: www.shaunreed.com ##
## A custom bash script to configure vim with my preferred settings      ##
## Run as user with sudo within directory to store / stash .vimrc configs  ##
#####

#####

# Example of easy colorization using printf
GREEN=$(tput setaf 2)
RED=$(tput setaf 1)
UNDERLINE=$(tput smul)
NORMAL=$(tput sgr0)

# Script Reduced, lines removed

# Example of creating an array of strings to be passed to printf
welcome=( "\nEnter 1 to configure vim with the Klips repository, any other value to exit." \
"The up-to-date .vimrc config can be found here: https://github.com/shaunrd0/klips/tree/master/configs" \
"${RED}Configuring Vim with this tool will update / upgrade your packages${NORMAL}\n\n")

# Create a printf format and pass the entire array to it
# Will iterate through array, filling format provided with array contents
# Useful for printing / formatting lists, instructions, etc
printf '%b\n' "${welcome[@]}"

read cChoice

# Script Reduced, lines removed
```

[Full script](#)

Using the above method, you could easily create a single array containing multiple responses to related paths the script could take for a related option, and refer to the appropriate index of the array directly, instead of passing all of the contents of the array to the same format.

For more advanced formatting, read the below script carefully, and you will have a basic understanding of how printf can be used dynamically within scripts to provide consistent formatting.

```
#!/bin/bash

divider=====
divider=$divider$divider

header="\n %-10s %8s %10s %11s\n"
format=" %-10s %08d %10s %11.2f\n"

width=43

printf "$header" "ITEM NAME" "ITEM ID" "COLOR" "PRICE"

printf "%$width.${width}s\n" "$divider"

printf "$format" \
Triangle 13  red 20 \
Oval 204449 "dark blue" 65.656 \
Square 3145 orange .7

# https://linuxconfig.org/bash-printf-syntax-basics-with-examples
```

String Manipulation

In bash, there are useful features to handle manipulating strings. These strings may be in any format, but the examples below will use strings that refer to directories and files as examples, since this is a common scenario.

```
#!/bin/bash

local teststring="/home/kapper/Code/"

echo "${teststring#/home/}"    # Remove shortest subtring from left matching pattern `/home/` (outputs
kapper/Code)

echo "${teststring##*kapper/}" # Remove longest subtring from left matching pattern `*/` (outputs Code/)
echo "${teststring%/*}"        # Remove shortest subtring from right matching pattern `/*` (outputs
/home/kapper/Code)
```

```

echo "${teststring%%kapper/*}" # Remove longest subtring from right matching pattern `kapper/*` (outputs
/home/)

# Can be used to obtain file name
local file="/home/kapper/.vimrc"
echo ${file##*/} # (outputs .vimrc)

# Can be used to obtain or strip file extensions
local other_file="/home/kapper/image.jpg"
echo ${other_file##*/} # (outputs image.jpg)
echo ${other_file%.*} # (outputs .jpg)

# Can piggy-back string manipulation, though it gets hard to read quickly
echo ${${other_file##*/}%.*} # (outputs image)

# Can remove up to first appearance of a space character (or tab, newline, other whitespace)
local white_space="sometextbeforeaspace /system/path/"
echo ${white_space%%[:space:]} # (outputs /system/path)

```

Arrays

Arrays can be declared and initialized with the syntax below. This script will output the word `collection`

```

#!/bin/bash
local arr=(a collection of single words speparated by spaces will automatically be split into indexed array)
echo ${arr[1]}

```

Alternatively, arrays of strings that include spaces can be declared without splitting by simply using double-quotes. This script will output `This array has two elements`

```

#!/bin/bash
local arr=("This will not be split" "This array has two elements")
echo ${arr[1]}

```

Arrays can also be associative, if we also define names for each index when initializing. The script below will output `example`

```

#!/bin/bash
local arr=([zero]=some [one]=example [two]=array)

```

```
echo ${arr[one]}
```

In the script below, we use an array to search for the package coretemp sensor of our CPU.

```
#!/bin/bash
## Author: Shaun Reed | Contact: shaunrd0@gmail.com | URL: www.shaunreed.com ##
##                                     ##
## A script to find and return the CPU package temp sensor          ##
#####
#####
# bash.sh

for i in /sys/class/hwmon/hwmon*/temp*_input; do
    # Append each sensor to an array
    sensors+=("${(<$(dirname $i)/name): $(cat ${i%_*}_label 2>/dev/null || echo $(basename ${i%_*}))
$(readlink -f $i)");
done

# Loop through initialized array of hardware temp sensors
for i in "${sensors[@]}"
do
    # If the sensor is for the CPU core package temp, export to env variable
    if [[ $i =~ ^coretemp:.Package.* ]]
    then
        export CPU_SENSOR=${i#*0}
    fi
done

# Convert from C to F using our exported CPU_SENSOR variable
echo "scale=2;((9/5) * $(cat $CPU_SENSOR)/1000) + 32"|bc
```

While Loops

While loop using conditionals within bash to automate cmake builds -

```
#!/bin/bash
## Author: Shaun Reed | Contact: shaunrd0@gmail.com | URL: www.shaunreed.com ##
## A custom bash script for building cmake projects.                ##
## Intended to be ran in root directory of the project alongside CMakeLists ##
#####
```

```
#####
```

```
# Infinite while loop - break on conditions
```

```
while true
```

```
do
```

```
    printf "\nEnter 1 to build, 2 to cleanup previous build, 0 to exit.\n"
```

```
    read bChoice
```

```
    # Build loop
```

```
    # If input read is == 1
```

```
    if [ $bChoice -eq 1 ]
```

```
    then
```

```
        mkdir build
```

```
        # Move to a different directory within a subshell and build the project
```

```
        # The '(' and ')' here preserves our working directory
```

```
        (cd build && cmake .. && cmake --build .)
```

```
    fi
```

```
    # Clean-up loop
```

```
    # If input read is == 2
```

```
    if [ $bChoice -eq 2 ]
```

```
    then
```

```
        printf "test\n"
```

```
        rm -Rv build/*
```

```
    fi
```

```
    # Exit loops, all other input -
```

```
    # If input read is >= 3, exit
```

```
    if [ $bChoice -ge 3 ]
```

```
    then
```

```
        break
```

```
    fi
```

```
    # If input read is <= 0, exit
```

```
    if [ $bChoice -le 0 ]
```

```
    then
```

```
        break
```

```
    fi
```

```
# Bash will print an error if symbol or character input
```

```
done
```

Subshells

Sometimes, you may want to stay in your active directory but perform some action in another. To do this, we can do something in a subshell (a background shell) -

```
$ (cd /var/log && cp -- *.log ~/Desktop)
```

Examples

Check out my snippet repository for more up-to-date scripts, configurations - [/shaunrd0/klips](#)

Basic script example of adding a user to a Linux system. This script uses parameters, basic formatting, and can either be ran as root or as a user without sudo, depending on the need. If needed, see the Knoats Book [Adding Linux Users](#) for more explanation on the below.

```
#!/bin/bash
## Author: Shaun Reed | Contact: shaunrd0@gmail.com | URL: www.shaunreed.com ##
## A custom bash script for creating new linux users. ##
## Syntax: ./adduser.sh <username> <userID> ##
#####

#####

if [ "$#" -ne 2 ]; then
    printf "Illegal number of parameters."
    printf "\nUsage: sudo ./adduser.sh <username> <groupid>"
    printf "\n\nAvailable groupd IDs:"
    printf "\n60001.....61183  Unused | 65520.....65533  Unused"
    printf "\n65536.....524287  Unused | 1879048191.....2147483647  Unused\n"
    exit
fi

sudo adduser $1 --gecos "First Last,RoomNumber,WorkPhone,HomePhone" --disabled-password --uid $2

printf "\nEnter 1 if $1 should have sudo privileges. Any other value will continue and make no changes\n"
read choice
if [ $choice -eq 1 ]; then
    printf "\nConfiguring sudo for $1...\n"
```



```
    sudo usermod -G sudo $1
fi

printf "\nEnter 1 to set a password for $1, any other value will exit with no password set\n"
read choice

if [ $choice -eq 1 ] ; then
    printf "\nChanging password for $1...\n"
    sudo passwd $1
fi
```

Revision #14

Created 21 July 2019 07:06:09 by Shaun Reed

Updated 27 January 2022 17:19:58 by Shaun Reed