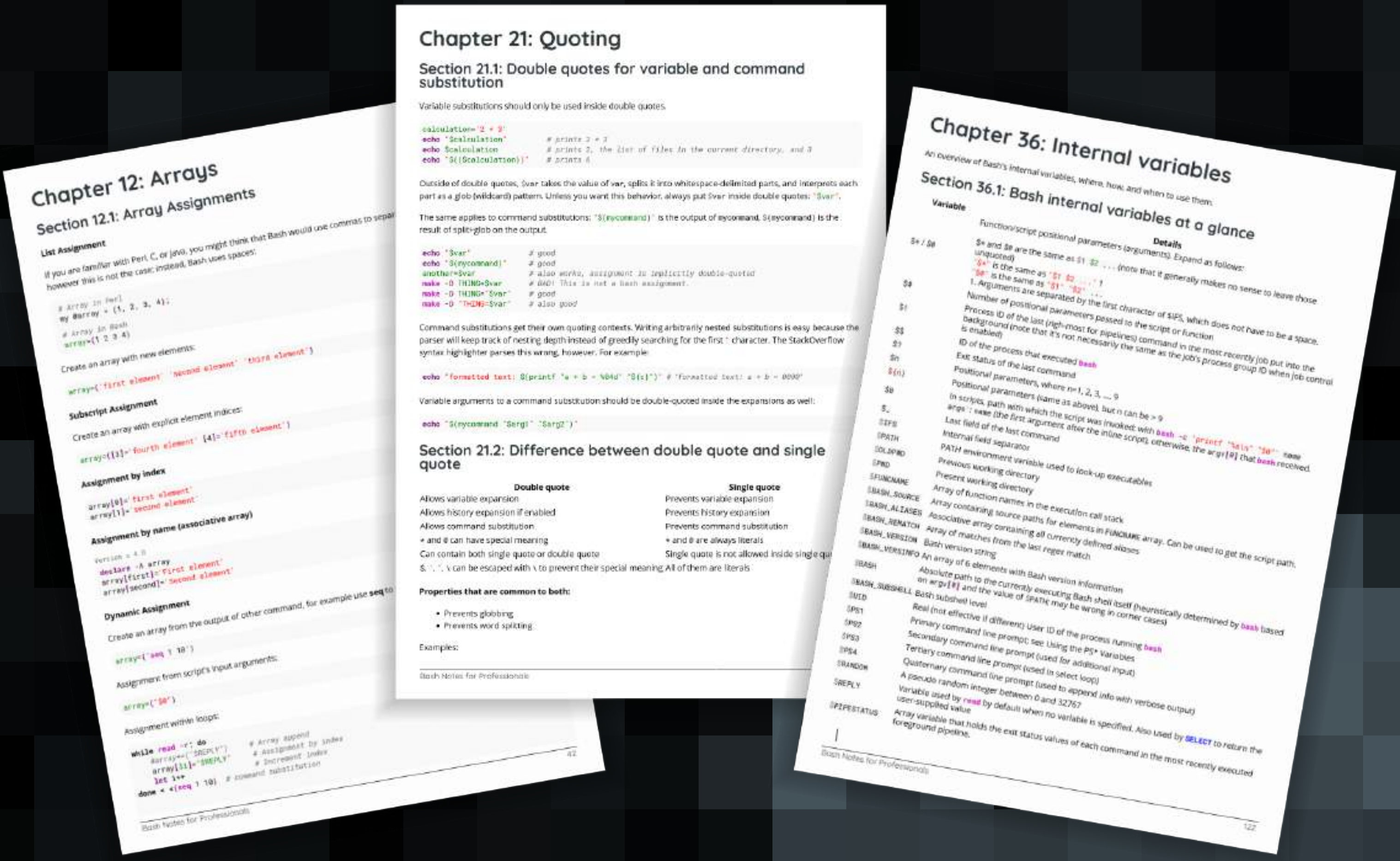
Bash

Notes for Professionals



Chapter 12: Arrays

Section 12.1: Array Assignments

List Assignment

If you are familiar with Perl, C, or Java, you might think that Bash would use commas to separ
however this is not the case; instead, Bash uses spaces:

```
# Array in Perl
my @array = (1, 2, 3, 4);
# Array in Bash
array=(1 2 3 4)
```

Create an array with new elements:

```
array=('first element' 'second element' 'third element')
```

Subscript Assignment

Create an array with explicit element indices:

```
array=([3]='fourth element' [4]='fifth element')
```

Assignment by index

```
array[0]='first element'
array[1]='second element'
```

Assignment by name (associative array)

```
Version ≥ 4.0
declare -A array
array[first]='First element'
array[second]='Second element'
```

Dynamic Assignment

Create an array from the output of other command, for example use seq to

```
array=(`seq 1 10`)
```

Assignment from script's input arguments:

```
array=("$@")
```

Assignment within loops:

```
while read -r; do
    #array+=("$REPLY")     # Array append
    array[$i]="$REPLY"     # Assignment by index
    let i++                # Increment index
done < <(seq 1 10)  # command substitution
```

Bash Notes for Professionals 42

Chapter 21: Quoting

Section 21.1: Double quotes for variable and command substitution

Variable substitutions should only be used inside double quotes.

```
calculation='2 * 3'
echo "$calculation"         # prints 2 * 3
echo $calculation           # prints 2, the list of files in the current directory, and 3
echo "$(($calculation))"    # prints 6
```

Outside of double quotes, $var takes the value of var, splits it into whitespace-delimited parts, and interprets each part as a glob (wildcard) pattern. Unless you want this behavior, always put $var inside double quotes: "$var".

The same applies to command substitutions: "$(mycommand)" is the output of mycommand, $(mycommand) is the result of split+glob on the output.

```
echo "$var"             # good
echo "$(mycommand)"     # good
another=$var            # also works, assignment is implicitly double-quoted
make -D THING=$var      # BAD! This is not a bash assignment.
make -D THING="$var"    # good
make -D "THING=$var"    # also good
```

Command substitutions get their own quoting contexts. Writing arbitrarily nested substitutions is easy because the parser will keep track of nesting depth instead of greedily searching for the first " character. The StackOverflow syntax highlighter parses this wrong, however. For example:

```
echo "formatted text: $(printf "a + b = %04d" "${c}")" # "formatted text: a + b = 0000"
```

Variable arguments to a command substitution should be double-quoted inside the expansions as well:

```
echo "$(mycommand "$arg1" "$arg2")"
```

Section 21.2: Difference between double quote and single quote

Double quote	Single quote
Allows variable expansion	Prevents variable expansion
Allows history expansion if enabled	Prevents history expansion
Allows command substitution	Prevents command substitution
* and @ can have special meaning	* and @ are always literals
Can contain both single quote or double quote	Single quote is not allowed inside single qu
$, `, ", \ can be escaped with \ to prevent their special meaning	All of them are literals

Properties that are common to both:

- Prevents globbing
- Prevents word splitting

Examples:

Bash Notes for Professionals

Chapter 36: Internal variables

An overview of Bash's internal variables, where, how, and when to use them.

Section 36.1: Bash internal variables at a glance

Variable	Details
$* / $@	Function/script positional parameters (arguments). Expand as follows: $* and $@ are the same as $1 $2 ... (note that it generally makes no sense to leave those unquoted) "$*" is the same as "$1 $2 ..." "$@" is the same as "$1" "$2" ... 1. Arguments are separated by the first character of $IFS, which does not have to be a space.
$#	Number of positional parameters passed to the script or function
$!	Process ID of the last (right-most for pipelines) command in the most recently job put into the background (note that it's not necessarily the same as the job's process group ID when job control is enabled)
$$	ID of the process that executed bash
$?	Exit status of the last command
$n	Positional parameters, where n=1, 2, 3, ..., 9
${n}	Positional parameters (same as above), but n can be > 9
$0	In scripts, path with which the script was invoked; with bash -c 'printf "%s\n" "$0"' name args': name (the first argument after the inline script), otherwise, the argv[0] that bash received.
$_	Last field of the last command
$IFS	Internal field separator
$PATH	PATH environment variable used to look-up executables
$OLDPWD	Previous working directory
$PWD	Present working directory
$FUNCNAME	Array of function names in the execution call stack
$BASH_SOURCE	Array containing source paths for elements in FUNCNAME array. Can be used to get the script path.
$BASH_ALIASES	Associative array containing all currently defined aliases
$BASH_REMATCH	Array of matches from the last regex match
$BASH_VERSION	Bash version string
$BASH_VERSINFO	An array of 6 elements with Bash version information
$BASH	Absolute path to the currently executing Bash shell itself (heuristically determined by bash based on argv[0] and the value of $PATH; may be wrong in corner cases)
$BASH_SUBSHELL	Bash subshell level
$UID	Real (not effective if different) User ID of the process running bash
$PS1	Primary command line prompt; see Using the PS* Variables
$PS2	Secondary command line prompt (used for additional input)
$PS3	Tertiary command line prompt (used in select loop)
$PS4	Quaternary command line prompt (used to append info with verbose output)
$RANDOM	A pseudo random integer between 0 and 32767
$REPLY	Variable used by read by default when no variable is specified. Also used by SELECT to return the user-supplied value
$PIPESTATUS	Array variable that holds the exit status values of each command in the most recently executed foreground pipeline.

Bash Notes for Professionals 122

100+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Contents

About

Chapter 1: Getting started with Bash

Version	Release Date
0.99	1989-06-08
1.01	1989-06-23
2.0	1996-12-31
2.02	1998-04-20
2.03	1999-02-19
2.04	2001-03-21
2.05b	2002-07-17
3.0	2004-08-03
3.1	2005-12-08
3.2	2006-10-11
4.0	2009-02-20
4.1	2009-12-31
4.2	2011-02-13
4.3	2014-02-26
4.4	2016-09-15

Section 1.1: Hello World

Interactive Shell

The Bash shell is commonly used **interactively:** It lets you enter and edit commands, then executes them when you press the `Return` key. Many Unix-based and Unix-like operating systems use Bash as their default shell (notably Linux and macOS). The terminal automatically enters an interactive Bash shell process on startup.

Output `Hello World` by typing the following:

```
echo "Hello World"
#> Hello World  # Output Example
```

Notes

- You can change the shell by just typing the name of the shell in terminal. For example: `sh`, **bash**, etc.
- **echo** is a Bash builtin command that writes the arguments it receives to the standard output. It appends a newline to the output, by default.

Non-Interactive Shell

The Bash shell can also be run **non-interactively** from a script, making the shell require no human interaction. Interactive behavior and scripted behavior should be identical – an important design consideration of Unix V7 Bourne shell and transitively Bash. Therefore anything that can be done at the command line can be put in a script file for reuse.

Follow these steps to create a `Hello World` script:

1. Create a new file called `hello-world.sh`

```
touch hello-world.sh
```

2. Make the script executable by running `chmod +x hello-world.sh`
3. Add this code:

   ```
   #!/bin/bash
   echo "Hello World"
   ```

   Line 1: The first line of the script must start with the character sequence `#!`, referred to as *shebang*1. The shebang instructs the operating system to run `/bin/bash`, the Bash shell, passing it the script's path as an argument.

   E.g. `/bin/bash hello-world.sh`

   Line 2: Uses the `echo` command to write `Hello World` to the standard output.

4. Execute the `hello-world.sh` script from the command line using one of the following:

   - `./hello-world.sh` – most commonly used, and recommended
   - `/bin/bash hello-world.sh`
   - `bash hello-world.sh` – assuming `/bin` is in your `$PATH`
   - `sh hello-world.sh`

For real production use, you would omit the `.sh` extension (which is misleading anyway, since this is a Bash script, not a `sh` script) and perhaps move the file to a directory within your `PATH` so that it is available to you regardless of your current working directory, just like a system command such as `cat` or `ls`.

Common mistakes include:

1. Forgetting to apply execute permission on the file, i.e., `chmod +x hello-world.sh`, resulting in the output of `./hello-world.sh: Permission denied`.
2. Editing the script on Windows, which produces incorrect line ending characters that Bash cannot handle.

   A common symptom is `: command not found` where the carriage return has forced the cursor to the beginning of line, overwriting the text before the colon in the error message.

   The script can be fixed using the `dos2unix` program.

   An example use: `dos2unix hello-world.sh`

   `dos2unix` edits the file inline.

3. Using `sh ./hello-world.sh`, not realizing that `bash` and `sh` are distinct shells with distinct features (though since Bash is backwards-compatible, the opposite mistake is harmless).

   Anyway, simply relying on the script's shebang line is vastly preferable to explicitly writing `bash` or `sh` (or `python` or `perl` or `awk` or `ruby` or...) before each script's file name.

   A common shebang line to use in order to make your script more portable is to use `#!/usr/bin/env bash` instead of hard-coding a path to Bash. That way, `/usr/bin/env` has to exist, but beyond that point, `bash` just needs to be on your PATH. On many systems, `/bin/bash` doesn't exist, and you should use `/usr/local/bin/bash` or some other absolute path; this change avoids having to figure out the details of that.

1 *Also referred to as sha-bang, hashbang, pound-bang, hash-pling.*

Section 1.2: Hello World Using Variables

Create a new file called `hello.sh` with the following content and give it executable permissions with `chmod +x hello.sh`.

> Execute/Run via: `./hello.sh`

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

# Note that spaces cannot be used around the `=` assignment operator
whom_variable="World"

# Use printf to safely output the data
printf "Hello, %s\n" "$whom_variable"
#> Hello, World
```

This will print `Hello, World` to standard output when executed.

To tell bash where the script is you need to be very specific, by pointing it to the containing directory, normally with `./` if it is your working directory, where `.` is an alias to the current directory. If you do not specify the directory, `bash` tries to locate the script in one of the directories contained in the `$PATH` environment variable.

The following code accepts an argument $1, which is the first command line argument, and outputs it in a formatted string, following `Hello,`.

> Execute/Run via: `./hello.sh World`

```
#!/usr/bin/env bash
printf "Hello, %s\n" "$1"
#> Hello, World
```

It is important to note that $1 has to be quoted in double quote, not single quote. `"$1"` expands to the first command line argument, as desired, while `'$1'` evaluates to literal string $1.

> **Security Note:**
> Read **Security implications of forgetting to quote a variable in bash shells** to understand the importance of placing the variable text within double quotes.

Section 1.3: Hello World with User Input

The following will prompt a user for input, and then store that input as a string (text) in a variable. The variable is then used to give a message to the user.

```
#!/usr/bin/env bash
echo  "Who are you?"
read name
echo "Hello, $name."
```

The command `read` here reads one line of data from standard input into the variable `name`. This is then referenced using `$name` and printed to standard out using `echo`.

Example output:

```
$ ./hello_world.sh
Who are you?
Matt
Hello, Matt.
```

Here the user entered the name "Matt", and this code was used to say `Hello, Matt.`.

And if you want to append something to the variable value while printing it, use curly brackets around the variable name as shown in the following example:

```
#!/usr/bin/env bash
echo  "What are you doing?"
read action
echo "You are ${action}ing."
```

Example output:

```
$ ./hello_world.sh
What are you doing?
Sleep
You are Sleeping.
```

Here when user enters an action, "ing" is appended to that action while printing.

Section 1.4: Importance of Quoting in Strings

Quoting is important for string expansion in bash. With these, you can control how the bash parses and expands your strings.

There are two types of quoting:

- **Weak**: *uses double quotes: "*
- **Strong**: *uses single quotes: '*

If you want to bash to expand your argument, you can use **Weak Quoting**:

```
#!/usr/bin/env bash
world="World"
echo "Hello $world"
#> Hello World
```

If you don't want to bash to expand your argument, you can use **Strong Quoting**:

```
#!/usr/bin/env bash
world="World"
echo 'Hello $world'
```

```
#> Hello $world
```

You can also use escape to prevent expansion:

```
#!/usr/bin/env bash
world="World"
echo "Hello \$world"
#> Hello $world
```

For more detailed information other than beginner details, you can continue to read it here.

Section 1.5: Viewing information for Bash built-ins

```
help <command>
```

This will display the Bash help (manual) page for the specified built-in.

For example, `help unset` will show:

> ```
> unset: unset [-f] [-v] [-n] [name ...]
> Unset values and attributes of shell variables and functions.
>
> For each NAME, remove the corresponding variable or function.
>
> Options:
> -f    treat each NAME as a shell function
> -v    treat each NAME as a shell variable
> -n    treat each NAME as a name reference and unset the variable itself
> rather than the variable it references
>
> Without options, unset first tries to unset a variable, and if that fails,
> tries to unset a function.
>
> Some variables cannot be unset; also see `readonly'.
>
> Exit Status:
> Returns success unless an invalid option is given or a NAME is read-only.
> ```

To see a list of all built-ins with a short description, use

```
help -d
```

Section 1.6: Hello World in "Debug" mode

```
$ cat hello.sh
#!/bin/bash
echo "Hello World"
$ bash -x hello.sh
+ echo Hello World
Hello World
```

The `-x` argument enables you to walk through each line in the script. One good example is here:

```
$ cat hello.sh
#!/bin/bash
echo "Hello World\n"
```

```
adding_string_to_number="s"
v=$(expr 5 + $adding_string_to_number)

$ ./hello.sh
Hello World

expr: non-integer argument
```

The above prompted error is not enough to trace the script; however, using the following way gives you a better sense where to look for the error in the script.

```
$ bash -x hello.sh
+ echo Hello World\n
Hello World

+ adding_string_to_number=s
+ expr 5 + s
expr: non-integer argument
+ v=
```

Section 1.7: Handling Named Arguments

```
#!/bin/bash

deploy=false
uglify=false

while (( $# > 1 )); do case $1 in
   --deploy) deploy="$2";;
   --uglify) uglify="$2";;
   *) break;
 esac; shift 2
done

$deploy && echo "will deploy... deploy = $deploy"
$uglify && echo "will uglify... uglify = $uglify"

# how to run
# chmod +x script.sh
# ./script.sh --deploy true --uglify false
```

Chapter 2: Script shebang

Section 2.1: Env shebang

To execute a script file with the **bash** executable found in the PATH environment variable by using the executable **env**, the **first line** of a script file must indicate the absolute path to the **env** executable with the argument **bash**:

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

The **env** path in the shebang is resolved and used only if a script is directly launch like this:

```
script.sh
```

The script must have execution permission.

The shebang is ignored when a **bash** interpreter is explicitly indicated to execute a script:

```
bash script.sh
```

Section 2.2: Direct shebang

To execute a script file with the **bash** interpreter, the **first line** of a script file must indicate the absolute path to the **bash** executable to use:

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

The **bash** path in the shebang is resolved and used only if a script is directly launch like this:

```
./script.sh
```

The script must have execution permission.

The shebang is ignored when a **bash** interpreter is explicitly indicated to execute a script:

```
bash script.sh
```

Section 2.3: Other shebangs

There are two kinds of programs the kernel knows of. A binary program is identified by it's ELF (**E**xtenable**L**oadable**F**ormat) header, which is usually produced by a compiler. The second one are scripts of any kind.

If a file starts in the very first line with the sequence **#!** then the next string has to be a pathname of an interpreter. If the kernel reads this line, it calls the interpreter named by this pathname and gives all of the following words in this line as arguments to the interpreter. If there is no file named "something" or "wrong":

```
#!/bin/bash something wrong
echo "This line never gets printed"
```

bash tries to execute its argument "something wrong" which doesn't exist. The name of the script file is added too. To see this clearly use an **echo** shebang:

```
#"/bin/echo something wrong
# and now call this script named "thisscript" like so:
# thisscript one two
# the output will be:
something wrong ./thisscript one two
```

Some programs like **awk** use this technique to run longer scripts residing in a disk file.

Chapter 3: Navigating directories

Section 3.1: Absolute vs relative directories

To change to an absolutely specified directory, use the entire name, starting with a slash `/`, thus:

```
cd /home/username/project/abc
```

If you want to change to a directory near your current on, you can specify a relative location. For example, if you are already in `/home/username/project`, you can enter the subdirectory `abc` thus:

```
cd abc
```

If you want to go to the directory above the current directory, you can use the alias `..`. For example, if you were in `/home/username/project/abc` and wanted to go to `/home/username/project`, then you would do the following:

```
cd ..
```

This may also be called going "up" a directory.

Section 3.2: Change to the last directory

For the current shell, this takes you to the previous directory that you were in, no matter where it was.

```
cd -
```

Doing it multiple times effectively "toggles" you being in the current directory or the previous one.

Section 3.3: Change to the home directory

The default directory is the home directory (`$HOME`, typically `/home/username`), so `cd` without any directory takes you there

```
cd
```

Or you could be more explicit:

```
cd $HOME
```

A shortcut for the home directory is `~`, so that could be used as well.

```
cd ~
```

Section 3.4: Change to the Directory of the Script

In general, there are two types of Bash **scripts**:

1. System tools which operate from the current working directory
2. Project tools which modify files relative to their own place in the files system

For the second type of scripts, it is useful to change to the directory where the script is stored. This can be done with the following command:

```
cd "$(dirname "$(readlink -f "$0")")"
```

This command runs 3 commands:

1. `readlink -f "$0"` determines the path to the current script (`$0`)
2. `dirname` converts the path to script to the path to its directory
3. `cd` changes the current work directory to the directory it receives from `dirname`