Bash++

Nathan Warner

Computer Science
Northern Illinois University
United States

Contents

1 The Set Builtin 5

1.1 Syntax 5

1.2 Basic structure 5

1.2.1 Options 5

1.2.2 Long options 5

1.2.3 Special Symbols 5

1.2.4 Arguments 6

1.3 Examples 6

1.4 set - and set – 6

1.5 Commonly used options 7

2 The Shopt Builtin 9

2.1 Syntax 9

2.2 Commonly used options 9

3 Ansi escape codes 11

3.1 Basic Format 11

3.2 Commonly used escape codes in bash 11

3.2.1 Reset Formatting 11

3.2.2 Text Styles 11

3.2.3 Text Colors 11

3.2.4 Background Colors 12

4 Locale Env variables 13

4.1 Components of a Locale 13

4.2 Locale Environment Variables 13

4.3 LC_ALL=C 14

5 POSIX character classes and regex string matching 15

5.1 List of POSIX Character Classes 15

5.2 With Regex Matching 17

6 The tr command (translate) 18

6.1 Syntax 18

6.2 Common Options 18

6.3 Basic Example: Convert lowercase to uppercase 18

6.4 Basic Example: Deleting characters 18

6.5 Basic Example: Squeezing Characters 18

6.6 Basic Example: Complimenting characters 19

7 Export 20

8 Functions in shell scripts 21

8.1 Defining a Function 21

8.2 Calling a Function 21

8.3 Function Parameters 21

8.4 Returning Values 21

8.4.1 Using return for Status Codes 22

8.5 Local Variables 22

9 Functions on the command line 23

9.1 Undefining (unsetting) a command line function 23

9.2 List environment functions 23

10 Order of operations of ”commands” 24

11 The type builtin 25

11.1 Options 25

12 The Nuances of $@ and $* 26

12.1 Without Quotes 26

12.2 With Quotes 26

12.2.1 ”$@” 26

12.2.2 "$*" . . . 26

13 Substitution Operators 27

14 Patterns-Matching operators 28

14.1 Extended pattern matching with shopt extglob . . . 29

15 Here documents and strings 30

15.1 Here strings . . . 30

15.1.1 Syntax . . . 30

15.1.2 Example . . . 30

15.2 Here document . . . 30

15.2.1 Syntax . . . 30

15.2.2 Example . . . 30

15.2.3 Example . . . 31

16 Looping through characters in a string 32

16.1 Read while Here string . . . 32

16.2 Using grep . . . 32

16.3 C-style for loop . . . 32

16.4 Sequence loop . . . 33

17 Basic Calculator commad (bc) 34

17.1 Basic Use . . . 34

17.2 Exponentiation . . . 34

17.3 Floating-Point Arithmetic . . . 34

18 printf 35

18.1 Syntax . . . 35

18.2 Format Specifiers . . . 35

19 Exercise solutions 36

19.1 Exercise 1 . . . 36

19.1.1 Problem Statement . . . 36

19.1.2 Solution 1 . . . 36

19.1.3 Alternative solution . . . 37
19.2 Exercise 2 . . . 38
19.2.1 Problem Statement . . . 38
19.2.2 Solution . . . 38
19.2.3 Alternative solution . . . 39
19.3 Exercise 3 . . . 40
19.3.1 Problem statement . . . 40
19.3.2 Solution . . . 40
19.3.3 Alternate solution . . . 41

The Set Builtin

This builtin is so complicated that it deserves its own section. set allows you to change the values of shell options and set the positional parameters, or to display the names and values of shell variables.

1.1 Syntax

```
set [-abefhkmnptuvxBCEHPT] [-o option-name] [--] [-] [argument …]
set [+abefhkmnptuvxBCEHPT] [+o option-name] [--] [-] [argument …]
```

If no options or arguments are supplied, set displays the names and values of all shell variables and functions, sorted according to the current locale, in a format that may be reused as input for setting or resetting the currently-set variables. Read-only variables cannot be reset. In POSIX mode, only shell variables are listed.

When options are supplied, they set or unset shell attributes. Options, if specified, have the following meanings:

1.2 Basic structure

```
set [options] [--] [arguments]
```

1.2.1 Options

- **[-abefhkmnptuvxBCEHPT]:** These are single-character options prefixed with a - (dash).
- **[+abefhkmnptuvxBCEHPT]:** These are the same options prefixed with a + (plus), which can be used to disable the corresponding option.

1.2.2 Long options

- **[-o option-name]:** This is used to enable a long-form option by its name.
- **[+o option-name]:** This is used to disable a long-form option by its name.

1.2.3 Special Symbols

- **–:** This indicates the end of options. Any arguments following – are treated as positional parameters and not options.
- **-:** Ends the parsing of options without any following arguments and resets positional parameters.

1.2.4 Arguments

- **[argument ...]:** These are the positional parameters or arguments that follow the options.

1.3 Examples

- **Enable xtrace and errexit options:**

```
set -ex
```

- **Disable xtrace and errexit options:**

```
set +ex
```

- **Enable the pipefail option using its long name:**

```
set -o pipefail
```

- **Disable the pipefail option using its long name:**

```
set +o pipefail
```

- **Setting positional parameters:**

```
set -- arg1 arg2 arg3
```

After this command, $1 will be arg1, $2 will be arg2, and $3 will be arg3.

1.4 set - and set –

```
# Before resetting, setting positional parameters
set -- foo bar
echo $1  # Outputs: foo
echo $2  # Outputs: bar

# Resetting positional parameters
set -
echo $1  # Outputs: (empty)
echo $2  # Outputs: (empty)
```

In this example, the - resets the positional parameters, effectively setting $1, $2, etc., to empty.

1.5 Commonly used options

beginitemize

`-e (errexit)`

- **Description**: Exit immediately if a command exits with a non-zero status.
- **Usage**: `set -e`
- **Example**: Useful in scripts to stop execution if any command fails, ensuring errors are caught early.

`-u (nounset)`

- **Description**: Treat unset variables as an error when substituting.
- **Usage**: `set -u`
- **Example**: Helps catch typos and other errors by ensuring all variables are defined.

`-x (xtrace)`

- **Description**: Print commands and their arguments as they are executed.
- **Usage**: `set -x`
- **Example**: Useful for debugging scripts by showing the flow of execution and values of variables.

`-o pipefail`

- **Description**: Return the exit status of the last command in the pipeline that failed.
- **Usage**: `set -o pipefail`
- **Example**: Ensures that a failure in any part of a pipeline is detected.

`-n (noexec)`

- **Description**: Read commands but do not execute them.
- **Usage**: `set -n`
- **Example**: Useful for checking the syntax of a script without executing it.

`-v (verbose)`

- **Description**: Print shell input lines as they are read.
- **Usage**: `set -v`
- **Example**: Helps in debugging by showing the script's input as it is being read.

`-f (noglob)`

- **Description**: Disable filename expansion (globbing).
- **Usage**: `set -f`
- **Example**: Prevents wildcard characters from being expanded into filenames.

`-a (allexport)`

- **Description**: Automatically export all variables to the environment.

- **Usage**: `set -a`
- **Example**: Useful for ensuring all variables are available to child processes.

`-b (notify)`

- **Description**: Enable asynchronous notification of background job completion.
- **Usage**: `set -b`
- **Example**: Useful to be notified immediately when background jobs finish.

`-h (hashall)`

- **Description**: Enable the command hashing feature.
- **Usage**: `set -h`
- **Example**: Speeds up command lookup by storing the location of commands.

The Shopt Builtin

Concept 1: This builtin allows you to change additional shell optional behavior.
□

2.1 Syntax

```
shopt [-pqsu] [-o] [optname …]
```

Toggle the values of settings controlling optional shell behavior. The settings can be either those listed below, or, if the -o option is used, those available with the -o option to the set builtin command (see The Set Builtin). With no options, or with the -p option, a list of all settable options is displayed, with an indication of whether or not each is set; if optnames are supplied, the output is restricted to those options. The -p option causes output to be displayed in a form that may be reused as input. Other options have the following meanings:

- **-s:** Enable (set) each optname.
- **-u:** Disable (unset) each optname.
- **-q:** Suppresses normal output; the return status indicates whether the optname is set or unset. If multiple optname arguments are given with -q, the return status is zero if all optnames are enabled; non-zero otherwise.
- **-o:** Restricts the values of optname to be those defined for the -o option to the set builtin (see The Set Builtin).

2.2 Commonly used options

- `autocd`
  - **Description**: Change to a directory just by typing its name.
- `cdspell`
  - **Description**: Correct minor spelling errors in directory names during `cd`.
- `checkhash`
  - **Description**: Check that the command hashes are up to date before using them.
- `checkwinsize`
  - **Description**: Check the window size after each command and update `LINES` and `COLUMNS`.
- `cmdhist`
  - **Description**: Save multi-line commands as a single history entry.

- `dotglob`
  - **Description**: Include hidden files (starting with `.`) in filename expansions.
- `expand_aliases`
  - **Description**: Enable alias expansion.
- `extglob`
  - **Description**: Enable extended pattern matching features.
- `histappend`
  - **Description**: Append to the history file, rather than overwriting it.
- `histverify`
  - **Description**: Allow history substitution to be edited before execution.
- `hostcomplete`
  - **Description**: Enable hostname completion.
- `lithist`
  - **Description**: Save multi-line commands in the history with embedded newlines rather than using semicolons.
- `nocaseglob`
  - **Description**: Perform case-insensitive filename matching.
- `nullglob`
  - **Description**: Allow filename patterns that match no files to expand to a null string, rather than themselves.
- `progcomp`
  - **Description**: Enable programmable completion features.
- `promptvars`
  - **Description**: Enable the expansion of `${...}` in prompt strings.
- `sourcepath`
  - **Description**: Use the value of `$PATH` to find the directory containing the file supplied as an argument to the `source` builtin.

Ansi escape codes

Concept 2: ANSI escape codes are sequences of characters used to control formatting, color, and other output options on text terminals. These codes are defined by the ANSI (American National Standards Institute) standard. They are widely supported in terminal emulators and command-line interfaces.
□

3.1 Basic Format

```
\e[<parameters><command>
```

- **\e:** The escape character, which can be written as \033 or \x1b in some contexts.
- **[:** The CSI (Control Sequence Introducer) character.
- **<parameters>:** A sequence of one or more parameters separated by semicolons.
- **<command>:** A letter that specifies the action to take (e.g., m for text formatting).

3.2 Commonly used escape codes in bash

3.2.1 Reset Formatting

- `\e[0m`: Reset all attributes to their defaults.

3.2.2 Text Styles

- `\e[1m`: Bold text.
- `\e[4m`: Underlined text.
- `\e[7m`: Inverse text (swap background and foreground colors).

3.2.3 Text Colors

- `\e[30m`: Black text.
- `\e[31m`: Red text.
- `\e[32m`: Green text.
- `\e[33m`: Yellow text.
- `\e[34m`: Blue text.
- `\e[35m`: Magenta text.
- `\e[36m`: Cyan text.
- `\e[37m`: White text.

3.2.4 Background Colors

- `\e[40m`: Black background.
- `\e[41m`: Red background.
- `\e[42m`: Green background.
- `\e[43m`: Yellow background.
- `\e[44m`: Blue background.
- `\e[45m`: Magenta background.
- `\e[46m`: Cyan background.
- `\e[47m`: White background.

Locale Env variables

Concept 3: A locale in Linux and Bash defines a set of parameters that tailor the environment to a specific language, region, or cultural preference. These parameters include settings for character encoding, date and time formats, numeric formats, collation (sort order), and more. Locales help ensure that software behaves correctly for users in different regions and with different languages.
□

4.1 Components of a Locale

A locale typically consists of several categories, each controlling a different aspect of the environment:

- **LC_CTYPE:** Character classification and case conversion. This includes settings for character encoding (e.g., UTF-8).
- **LC_NUMERIC:** Numeric formatting, such as the decimal point and thousands separator.
- **LC_TIME:** Date and time formatting.
- **LC_COLLATE:** String collation (sorting order).
- **LC_MONETARY:** Monetary formatting, such as the currency symbol and decimal places.
- **LC_MESSAGES:** Localization of messages and responses (e.g., ”yes” and ”no”).
- **LC_PAPER:** Paper size settings.
- **LC_NAME:** Name format.
- **LC_ADDRESS:** Address format.
- **LC_TELEPHONE:** Telephone number format.
- **LC_MEASUREMENT:** Measurement units (e.g., metric vs. imperial).
- **LC_IDENTIFICATION:** Metadata about the locale itself.

4.2 Locale Environment Variables

Several environment variables control the locale settings in Linux and Bash:

- **LANG:** Sets the default locale for all categories unless overridden by more specific LC_* variables.
- **LC_ALL:** Overrides all other locale settings, ensuring a uniform locale for all categories.
- **LC_CTYPE, LC_NUMERIC, LC_TIME, etc.:** Set the locale for specific categories.

4.3 LC_ALL=C

- **LC_ALL:** This environment variable overrides all other locale settings. When it is set, it ensures that all aspects of the locale (such as character encoding, collation, date/time formats, etc.) use the specified locale value.
- **C:** The C locale, also known as the POSIX locale, is a standard locale that provides a consistent environment across all systems. It uses the ASCII character set and provides predictable, straightforward behavior.

Setting LC_ALL=C ensures that all locale-related operations in the script use the C locale. This can be useful for ensuring consistent behavior regardless of the user's local settings, especially for text processing, sorting, and other locale-sensitive operations.

POSIX character classes and regex string matching

Concept 4: POSIX character classes are a set of predefined character classes used in regular expressions to match specific types of characters. They are defined by the POSIX (Portable Operating System Interface) standard and provide a convenient way to specify common character types in a portable and readable manner.
□

5.1 List of POSIX Character Classes

- `[:alnum:]`
  - Matches any alphanumeric character, which includes both letters and digits.
  - Equivalent to `[A-Za-z0-9]`.
- `[:alpha:]`
  - Matches any alphabetic character, which includes both uppercase and lowercase letters.
  - Equivalent to `[A-Za-z]`.
- `[:blank:]`
  - Matches any blank character, which includes spaces and tabs.
  - Equivalent to `[ \t]`.
- `[:cntrl:]`
  - Matches any control character. These are non-printable characters in the ASCII range 0–31 and 127.
  - Examples include newline (`\n`), carriage return (`\r`), and escape (`\e`).
- `[:digit:]`
  - Matches any digit.
  - Equivalent to `[0-9]`.
- `[:graph:]`
  - Matches any printable character except for spaces. This includes punctuation, digits, and letters.
  - Equivalent to `[∧[:space:]]` but excluding space.
- `[:lower:]`
  - Matches any lowercase letter.
  - Equivalent to `[a-z]`.

- `[:print:]`
  - Matches any printable character, including spaces. This includes punctuation, digits, letters, and space.
  - Equivalent to `[∧[:cntrl:]]`.
- `[:punct:]`
  - Matches any punctuation character. This includes symbols and punctuation marks but not spaces, letters, or digits.
  - Examples include `!`, `@`, `#`, `$`, etc.
- `[:space:]`
  - Matches any whitespace character. This includes spaces, tabs, newlines, carriage returns, vertical tabs, and form feeds.
  - Equivalent to `[ \t\r\n\v\f]`.
- `[:upper:]`
  - Matches any uppercase letter.
  - Equivalent to `[A-Z]`.
- `[:xdigit:]`
  - Matches any hexadecimal digit.
  - Equivalent to `[0-9A-Fa-f]`.

5.2 With Regex Matching

```
check_characters() {
    local str="$1"

    if [[ "$str" =~ [[:alnum:]] ]]; then
    echo "String contains alphanumeric characters."
    fi

    if [[ "$str" =~ [[:alpha:]] ]]; then
    echo "String contains alphabetic characters."
    fi

    if [[ "$str" =~ [[:blank:]] ]]; then
    echo "String contains blank characters (space or tab)."
    fi

    if [[ "$str" =~ [[:digit:]] ]]; then
    echo "String contains digits."
    fi

    if [[ "$str" =~ [[:lower:]] ]]; then
    echo "String contains lowercase letters."
    fi

    if [[ "$str" =~ [[:upper:]] ]]; then
    echo "String contains uppercase letters."
    fi

    if [[ "$str" =~ [[:punct:]] ]]; then
    echo "String contains punctuation characters."
    fi

    if [[ "$str" =~ [[:space:]] ]]; then
    echo "String contains whitespace characters."
    fi

    if [[ "$str" =~ [[:xdigit:]] ]]; then
    echo "String contains hexadecimal digits."
    fi
}
```

Note:-

Notice the extra set of brackets, this is because we put the Posix character class inside of a regex character class. This means we can use the outer set of brackets as a normal regex class, add negation symbol, other characters, etc

The tr command (translate)

The tr command in Unix and Unix-like operating systems is a utility for translating or deleting characters. It reads from standard input and writes to standard output. The tr command is commonly used in scripts and command-line operations to perform simple text transformations.

6.1 Syntax

```
tr [OPTION]... SET1 [SET2]
```

- **SET1:** The set of characters to be replaced or deleted.
- **SET2:** The set of characters to replace the characters in SET1 (if provided).

6.2 Common Options

- **-d:** Delete characters in SET1, do not translate.
- **-s:** Squeeze repeated characters in SET1 into a single character.
- **-c:** Complement the characters in SET1.

6.3 Basic Example: Convert lowercase to uppercase

```
echo "Hello World" | tr "[:lower:]" "[:upper:]"
```

6.4 Basic Example: Deleting characters

```
echo "Hello World" | tr -d "aeiou"
```

6.5 Basic Example: Squeezing Characters

```
echo "Hello     world" | tr -s " "
```

6.6 Basic Example: Complimenting characters

```
echo "abcd123efg" | tr -c "[:digit:]" "-"
```

Replaces all characters **except** digits with a hyphen:

Export

the export command is used to set environment variables that will be available to child processes. When you export a variable, it becomes part of the environment of subsequently executed commands, including scripts and other programs.

```bash
MY_VAR="Hello"
export MY_VAR
```

Or more concisely

```bash
export MY_VAR="Hello"
```

Once exported, MY_VAR is available to any child process started from this shell. For example, if you start a new shell or run a script, MY_VAR will be available:

```bash
bash -c 'echo $MY_VAR'
```

Functions in shell scripts

8.1 Defining a Function

The basic syntax for defining a function in Bash is:

```
function fn {
}

// Or...

fn() {

}
```

8.2 Calling a Function

Once a function is defined, you can call it by simply using its name:

```
fn
```

8.3 Function Parameters

Functions can take parameters, which are accessed using positional parameters $1, $2, etc.

```
function greet {
    local name="$1"
    echo "Hello, $name!"
}

# Calling the function with a parameter
greet "Alice"
```

8.4 Returning Values

Bash functions can return a status code using the return keyword. To return a value, you typically use echo and capture the output in a variable.

```bash
function add {
    local a="$1"
    local b="$2"
    echo $((a + b))
}

# Capturing the output of the function
result=$(add 3 5)
echo "The sum is $result"
```

8.4.1 Using return for Status Codes

```bash
function check_file {
    local file="$1"
    if [[ -e "$file" ]]; then
        return 0
    else
        return 1
    fi
}

# Checking the status code
check_file "somefile.txt"
if [[ $? -eq 0 ]]; then
    echo "File exists"
else
    echo "File does not exist"
fi
```

8.5 Local Variables

Using local inside a function makes the variable scope local to that function, preventing it from affecting other parts of the script.

```bash
function example {
    local var="I am local"
    echo "$var"
}
example
echo "$var"  # This will not print anything since var is local
↪  to the function
```

Functions on the command line

We can create functions on the command line similar to how we create them in shell scripts. The syntax we use here is

```
fn() { // Press enter here
    > // Line 1
    > // Line 2
} // Typing a closing brace and hitting return ends the function

// Or...

function fn { // Press enter here
    > // Line 1
    > // Line 2
} // Typing a closing brace and hitting return ends the function

// Or...
function fn() { // Press enter here
    > // Line 1
    > // Line 2
} // Typing a closing brace and hitting return ends the function
```

Where *fn* is some arbitrary name

9.1 Undefining (unsetting) a command line function

To delete a function definition from the environment, we use

```
unset -f function_name
```

9.2 List environment functions

We can list available functions with

```
declare -f // Names and definitions
declare -F // Names only
```

Order of operations of ”commands”

1. Aliases
2. Keywords such as **function**, *if*, etc...
3. Functions
4. Built-ins like *cd* and *type*
5. Scripts and executables

The type builtin

The type command in Bash is used to display information about the command type and how it would be interpreted if used. This includes determining whether a command is a built-in shell command, an alias, a function, or an executable file located in the system's PATH. It's a useful tool for understanding how Bash will execute a command and for debugging.

```
type command_name
```

```
Check for a Built-in Command // cd is a shell builtin
```

11.1 Options

- **-a**: will show all possible locations for a command:
- **-t**: This option makes type print a single word that indicates the type of the command
- **-p**: This option forces type to print the path of the executable, similar to the which command

The Nuances of $@ and $*

In Bash, $@ and $* are special variables that represent all the positional parameters (arguments) passed to a script or a function. While they might seem similar, they have distinct behaviors when quoted, which affects how they handle arguments.

12.1 Without Quotes

When used without quotes, both $@ and $* behave the same way, expanding to all the positional parameters separated by spaces:

12.2 With Quotes

The difference between $@ and $* becomes apparent when they are used within double quotes.

12.2.1 ”$@”

- ”$@” expands to a list of all positional parameters, where each parameter is quoted separately.
- This means that each argument is treated as a separate word.

12.2.2 ”$*”

- ”$*” expands to a single string where all the positional parameters are concatenated into a single word, separated by the first character of the IFS (Internal Field Separator) variable (by default a space).
- This means that all arguments are combined into one word.

Substitution Operators

- **${varname:-word}**: If varname exists and isn't null, return its value; otherwise return word
  - **Purpose:** Returning a default value if the variable is undefined
- **${varname:=word}**: If varname exists and isn't null, return its value; otherwise set it to word and then return its value.
  - **Purpose:** Setting a variable to a default value if it is undefined
- **${varname:?message}**: If varname exists and isn't null, return its value: otherwise print varname: followed by message, and abort the current command or script. Omitting message produces the default message null or not set
  - **Purpose:** Catching errors that result from variables being undefined
- **${varname:+word}**: If varname exists and isn't null, return word; otherwise return null
  - **Purpose:** testing for the existence of a variable
- **${varname:offset:length}**: Substring expansion
  - **Purpose:** Returning parts of a string

Patterns-Matching operators

- **${variablepattern}**: If the pattern matches the beginning of the variable's value, delete the shortest part that matches and return the rest.

```
path="/home/user/docs/report.txt"

# Remove the shortest match of pattern from the beginning
result=${path#*/} // Output: home/user/docs/report.txt
```

- **${variablepattern}**: If the pattern matches the beginning of the variable's value, delete the longest part that matches and return the rest

```
 path="/home/user/docs/report.txt"

# Remove the shortest match of pattern from the beginning
result=${path##*/} // Output: report.txt
```

- **${variable%pattern}**: If the pattern matches the end of the variable's value, delete the shortest part that matches and return the rest.

```
filename="document.txt.bak"

# Remove the shortest match of pattern from the end
result=${filename%.bak} // Output: document.txt
```

- **${variable%%pattern}**: If the pattern matches the end of the variable's value, delete the longest part that matches and return the rest.

```
filename="document.txt.bak"

# Remove the shortest match of pattern from the end
result=${filename%.bak} // Output: document
```

- **${variable/pattern/string}**: Replaces the first occurrence of pattern in variable with string.

```
text="Hello world, welcome to the world of Bash"

# Replace the first occurrence of "world" with "universe"
result=${text/world/universe} // Output: Hello universe, welcome to the world of Bash
```

- **${variable//pattern/string}**: Replaces the first occurrence of pattern in variable with string.

```
 text="Hello world, welcome to the world of Bash"

# Replace all occurrences of "world" with "universe"
result=${text//world/universe} // Output: Hello universe,
 ↪  welcome to the universe of Bash
```

14.1 Extended pattern matching with shopt extglob

Bash provides a further set of pattern matching operators if the **shopt** option **extglob** is switched on. Each operator takes one or more patterns, normally strings, separated by the vertical bar (|). The extended pattern matching operators are given below

- ***(patternlist)**: Matches zero or more occurrences of the given pattern
- **+(patternlist)**: Matches one or more occurrences of the given pattern
- **?(patternlist)**: Matches zero or one occurrences of the given pattern
- **@(patternlist)**: Matches exactly one of the given patterns
- **!(patternlist)**: Matches everything expect one of the given patterns

Here documents and strings

Here strings and here documents are both ways to provide input to commands in Bash, but they are used in different contexts and have different syntaxes and purposes.

15.1 Here strings

A here string allows you to pass a single line of text directly into the standard input (stdin) of a command. It is a simple and concise way to provide input for commands that expect input from stdin.

15.1.1 Syntax

```
command <<< "string"
```

15.1.2 Example

```
cat <<< "Hello, world!"
```

15.2 Here document

A here document allows you to pass multiple lines of text to the stdin of a command. It is more suited for providing larger blocks of text or scripts as input.

15.2.1 Syntax

```
command <<EOF
line1
line2
line3
EOF
```

15.2.2 Example

```
cat <<EOF
Hello,
world!
This is a multi-line input.
EOF
```

15.2.3 Example

```
cat <<EOF > config.txt
[server]
host = localhost
port = 8080

[client]
user = admin
EOF

cat config.txt
```

Looping through characters in a string

16.1 Read while Here string

```
main() {
    local inp=$1

    while IFS= read -r -n1 char; do
        echo "$char"
    done <<< "$inp"
}
main "$@"
```

16.2 Using grep

```
main() {
    local inp=$1

    for char in $(echo $inp | grep -o .); do
        echo "$char"
    done
}
main "$@"
```

16.3 C-style for loop

```
main() {
    local inp=$1

    for (( i=0; i<${#inp}; i++ )); do
        echo "${inp:i:1}"
    done
}
main "$@"
```

16.4 Sequence loop

```bash
main() {
    local inp=$1

    for i in $(seq 0 $((${#inp} -1))); do
        echo "${inp:i:1}"
    done


}
main "$@"
```

Basic Calculator commad (bc)

The bc command in Unix-like systems is an arbitrary precision calculator language, which is useful for performing mathematical operations that go beyond the capabilities of standard shell arithmetic. Here's a detailed explanation of the bc command, how it works, and some examples to illustrate its use.

17.1 Basic Use

You can use bc interactively by simply typing bc in the terminal, or you can use it non-interactively within a script or from the command line by echoing expressions into it.

```
echo "expression" | bc

echo "2+2" | bc
```

17.2 Exponentiation

We use the carrot ($\wedge$) for exponentiation

```
echo "2^64" | bc
```

17.3 Floating-Point Arithmetic

```
echo "scale=2; 5/3" | bc
```

Where *scale=2* Sets the number of decimal places to 2

printf

The printf command in Bash is a powerful tool for formatting and printing text. It is more flexible and powerful than the echo command because it allows you to specify the format of the output.

18.1 Syntax

```
printf <FORMAT> [arg]...

```

- **FORMAT:** A string that contains plain text and format specifiers.
- **ARGUMENT:** Values to be formatted according to the format specifiers.

18.2 Format Specifiers

- **%d:** Decimal integer.
- **%f:** Floating-point number.
- **%s:** String.
- **%x:** Hexadecimal number.
- **%o:** Octal number.

Exercise solutions

19.1 Exercise 1

19.1.1 Problem Statement

Calculate the Hamming Distance between two DNA strands.

Your body is made up of cells that contain DNA. Those cells regularly wear out and need replacing, which they achieve by dividing into daughter cells. In fact, the average human body experiences about 10 quadrillion cell divisions in a lifetime!

When cells divide, their DNA replicates too. Sometimes during this process mistakes happen and single pieces of DNA get encoded with the incorrect information. If we compare two strands of DNA and count the differences between them we can see how many mistakes occurred. This is known as the ”Hamming Distance”.

We read DNA using the letters C,A,G and T. Two strands might look like this:

GAGCCTACTAACGGGAT
CATCGTAATGACGGCCT

They have 7 differences, and therefore the Hamming Distance is 7.

19.1.2 Solution 1

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

main() {
    if [[ $# -ne 2 ]]; then
        echo "Usage: hamming.sh <string1> <string2>"
        exit 1
    fi

    if [[ ${#1} != ${#2} ]]; then
        echo "strands must be of equal length"
        exit 1
    fi

    counter=0;
    for (( i=0; i<${#1}; i++ )); do
        if [[ "${1:i:1}" != "${2:i:1}" ]]; then
            ((counter++)) // or -> counter=$((counter + 1))
        fi
    done

    echo $counter
}
main "${@}"
```

19.1.3 Alternative solution

```bash
#!/usr/bin/env bash
error () {
    printf '%s\n' "$*"
    exit 1
}
main () {
    (( $# == 2 )) || error 'Usage: hamming.sh <string1> <string2>'

    # Regular vars are easier to read when doing fancy parameter expansion.
    a=$1 b=$2

    # Using the a==b||... pattern everywhere in this function. I like consistency.
    (( ${#a} == ${#b} )) || error 'left and right strands must be of equal length'

    declare -i count
    for (( i = 0; i < ${#a}; i++ )); do
        [[ ${a:i:1} == "${b:i:1}" ]] || count+=1
    done

    printf '%d\n' "$count"
}
% main "$@"
```

19.2 Exercise 2

19.2.1 Problem Statement

Convert a phrase to its acronym.

Techies love their TLA (Three Letter Acronyms)!

Help generate some jargon by writing a program that converts a long name like Portable Network Graphics to its acronym (PNG).

Punctuation is handled as follows: hyphens are word separators (like whitespace); all other punctuation can be removed from the input.

Input	Output
As Soon As Possible	ASAP
Liquid-crystal display	LCD
Thank George It's Friday!	TGIF

19.2.2 Solution

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

main() {

    ac=""
    str=$(echo "$1" | sed 's/[^[:alnum:][:space:][:digit:]-]//g')

    [[ ${str:0:1} =~ [[:alpha:]] ]] && ac+=${str:0:1}

    for (( i=0; i<${#str}; i++ )); do
        if [[ ${str:i:1} =~ [[:space:]-] ]]  && [[ ${str:i+1:1} =~ [[:alpha:]] ]]; then
            ac+=$(echo ${str:i+1:1} | tr "[:lower:]" "[:upper:]")
        fi
    done

    echo $ac

}
main "${@}"
```

19.2.3 Alternative solution

```bash
#!/usr/bin/env bash
set -o errexit
set -o nounset
main() {
    local line="$1"
    IFS=' -_*' read -r -a words <<< "$line"
    local output=""
    local word
    for word in "${words[@]}"; do
        local letter="${word:0:1}"
        output+="${letter^^}"
    done
    echo "$output"
}
main "$@"
```

19.3 Exercise 3

19.3.1 Problem statement

Scrabble is a word game where players place letter tiles on a board to form words. Each letter has a value. A word's score is the sum of its letters' values.

Your task is to compute a word's Scrabble score by summing the values of its letters.

The letters are valued as follows:

Letter	Value
A, E, I, O, U, L, N, R, S, T	1
D, G	2
B, C, M, P	3
F, H, V, W, Y	4
K	5
J, X	8
Q, Z	10

For example, the word "cabbage" is worth 14 points:

19.3.2 Solution

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

main() {
    local count=0
    local inp=$(echo $1 | tr "[:upper:]" "[:lower:]")

    while IFS= read -r -n1 char; do
        if [[ $char =~ [aeioulnrst] ]]; then
            count=$(( $count + 1 ))
        elif [[ $char =~ [dg] ]]; then
            count=$(($count + 2))
         elif [[ $char =~ [bcmp] ]]; then
            count=$(($count + 3))
         elif [[ $char =~ [fhvwy] ]]; then
            count=$(($count + 4))
         elif [[ $char =~ [k] ]]; then
            count=$(($count + 5))
          elif [[ $char =~ [jx] ]]; then
            count=$(($count + 8))
        elif [[ $char =~ [qz] ]]; then
            count=$(($count + 10))
        fi
    done <<< "$inp"

    echo "$count"
}
main "$@"
```

19.3.3 Alternate solution

```bash
#!/usr/bin/env bash
total=0
for x in $(echo ${1^^} | grep -o .); do
    case $x in
        [AEIOULNRST]) ((total++));;
        [DG])         ((total+=2));;
        [BCMP])       ((total+=3));;
        [FHVWY])      ((total+=4));;
        K)            ((total+=5));;
        [JX])         ((total+=8));;
        *)            ((total+=10));;
    esac
done
echo $total
```