

Bash++

Nathan Warner



**Northern Illinois
University**

Computer Science
Northern Illinois University
United States

Contents

1	The Set Builtin	3
1.1	Syntax	3
1.2	Basic structure	3
1.2.1	Options	3
1.2.2	Long options	3
1.2.3	Special Symbols	3
1.2.4	Arguments	4
1.3	Examples	4
1.4	set - and set -	4
1.5	Commonly used options	5
2	The Shopt Builtin	7
2.1	Syntax	7
2.2	Commonly used options	7
3	Ansi escape codes	9
3.1	Basic Format	9
3.2	Commonly used escape codes in bash	9
3.2.1	Reset Formatting	9
3.2.2	Text Styles	9
3.2.3	Text Colors	9
3.2.4	Background Colors	10
4	Locale Env variables	11
4.1	Components of a Locale	11
4.2	Locale Environment Variables	11
4.3	LC_ALL=C	12

5	POSIX character classes and regex string matching	13
5.1	List of POSIX Character Classes	13
5.2	With Regex Matching	15
6	The tr command (translate)	16
6.1	Syntax	16
6.2	Common Options	16
6.3	Basic Example: Convert lowercase to uppercase	16
6.4	Basic Example: Deleting characters	16
6.5	Basic Example: Squeezing Characters	16
6.6	Basic Example: Complimenting characters	17
7	Exercise solutions	18
7.1	Exercise 1	18
7.1.1	Problem Statement	18
7.1.2	Solution 1	18
7.1.3	Alternative solution	19
7.2	Exercise 2	20
7.2.1	Problem Statement	20
7.2.2	Solution	20
7.2.3	Alternative solution	21

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

```
1  set [-abefhkmnptuvxBCEHPT] [-o option-name] [--] [-] [argument ...]
2  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

```
1  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:**

```
1 set -ex
```

- **Disable xtrace and errexit options:**

```
1 set +ex
```

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

```
1 set -o pipefail
```

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

```
1 set +o pipefail
```

- **Setting positional parameters:**

```
1 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 -

```
1 # Before resetting, setting positional parameters
2 set -- foo bar
3 echo $1 # Outputs: foo
4 echo $2 # Outputs: bar
5
6 # Resetting positional parameters
7 set -
8 echo $1 # Outputs: (empty)
9 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

```
1  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

```
1  \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

```
1  check_characters() {
2      local str="$1"
3
4      if [[ "$str" =~ [[:alnum:]] ]]; then
5          echo "String contains alphanumeric characters."
6      fi
7
8      if [[ "$str" =~ [[:alpha:]] ]]; then
9          echo "String contains alphabetic characters."
10     fi
11
12     if [[ "$str" =~ [[:blank:]] ]]; then
13         echo "String contains blank characters (space or tab)."
14     fi
15
16     if [[ "$str" =~ [[:digit:]] ]]; then
17         echo "String contains digits."
18     fi
19
20     if [[ "$str" =~ [[:lower:]] ]]; then
21         echo "String contains lowercase letters."
22     fi
23
24     if [[ "$str" =~ [[:upper:]] ]]; then
25         echo "String contains uppercase letters."
26     fi
27
28     if [[ "$str" =~ [[:punct:]] ]]; then
29         echo "String contains punctuation characters."
30     fi
31
32     if [[ "$str" =~ [[:space:]] ]]; then
33         echo "String contains whitespace characters."
34     fi
35
36     if [[ "$str" =~ [[:xdigit:]] ]]; then
37         echo "String contains hexadecimal digits."
38     fi
39 }
```

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

```
1 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

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

6.4 Basic Example: Deleting characters

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

6.5 Basic Example: Squeezing Characters

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

6.6 Basic Example: Complimenting characters

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

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

Exercise solutions

7.1 Exercise 1

7.1.1 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.

7.1.2 Solution 1

```
1  #!/usr/bin/env bash
2
3  main() {
4      if [[ $# -ne 2 ]]; then
5          echo "Usage: hamming.sh <string1> <string2>"
6          exit 1
7      fi
8
9      if [[ ${#1} != ${#2} ]]; then
10         echo "strands must be of equal length"
11         exit 1
12     fi
13
14     counter=0;
15     for (( i=0; i<${#1}; i++ )); do
16         if [[ "${1:i:1}" != "${2:i:1}" ]]; then
17             ((counter++)) // or -> counter=$((counter + 1))
18         fi
19     done
20
21     echo $counter
22 }
23 main "$@"
```

7.1.3 Alternative solution

```
1  #!/usr/bin/env bash
2  error () {
3      printf '%s\n' "$*"
4      exit 1
5  }
6  main () {
7      (( $# == 2 )) || error 'Usage: hamming.sh <string1>
↳ <string2>'
8
9      # Regular vars are easier to read when doing fancy parameter
↳ expansion.
10     a=$1 b=$2
11
12     # Using the a==b||... pattern everywhere in this function. I
↳ like consistency.
13     (( ${#a} == ${#b} )) || error 'left and right strands must
↳ be of equal length'
14
15     declare -i count
16     for (( i = 0; i < ${#a}; i++ )); do
17         [[ ${a:i:1} == "${b:i:1}" ]] || count+=1
18     done
19
20     printf '%d\n' "$count"
21 }
22 main "$@"
```

7.2 Exercise 2

7.2.1 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

7.2.2 Solution

```
1  #!/usr/bin/env bash
2
3  main() {
4
5      ac=""
6      str=$(echo "$1" | sed 's/[^[[:alnum:]][[:space:]][[:digit:]]-]//g')
7
8      [[ ${str:0:1} =~ [[:alpha:]] ]] && ac+=${str:0:1}
9
10     for (( i=0; i<${#str}; i++ )); do
11         if [[ ${str:i:1} =~ [[:space:]]- ] ] && [[ ${str:i+1:1}
↪  =~ [[:alpha:]] ] ]; then
12             ac+=$(echo ${str:i+1:1} | tr "[:lower:]" "[:upper:]")
13         fi
14     done
15
16     echo $ac
17
18 }
19 main "$@"
```

7.2.3 Alternative solution

```
1  #!/usr/bin/env bash
2  set -o errexit
3  set -o nounset
4  main() {
5      local line="$1"
6      IFS=' _*' read -r -a words <<< "$line"
7      local output=""
8      local word
9      for word in "${words[@]}; do
10         local letter="${word:0:1}"
11         output+="${letter^^}"
12     done
13     echo "$output"
14 }
15 main "$@"
```