Bash++

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

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

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

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

The reverse (rev) command

7.1 Reversing Strings

To reverse a string, we can use the rev command. The rev command words similar to the bc command

```
echo "string" | rev
```

7.2 Reversing arrays

```
declare -a arr1=("a" "b" "c")
declare -a arr2=($(echo "${arr1[@]}" | rev))

echo "${arr1[@]}" // Output: a b c
echo "${arr2[@]}" // Output: c b a
```

The fold command

The fold command in Unix-like operating systems is used to wrap each input line to fit within a specified width. It's particularly useful for breaking long lines of text into shorter, more manageable pieces. By default, fold breaks lines at a width of 80 characters, but this can be adjusted using options.

8.1 Syntax

```
fold [OPTION]... [FILE]...
```

8.2 Common Options

- **-w, --width=WIDTH:** Specify the maximum line width. Lines longer than this width will be broken.
- **-s, --spaces:** Break lines at word boundaries (spaces) instead of exactly at the specified width.

8.3 Example

```
echo "This is a long line that needs to be wrapped" | fold -w 20
```

8.4 Example

```
str="Helloworld"
str=$(fold -w 5 <<< $str)

echo $str // Output: Hello world
```

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.

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

Or more concisely

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

Functions in shell scripts

10.1 Defining a Function

The basic syntax for defining a function in Bash is:

```
function fn {
}

// Or...

fn() {

}
```

10.2 Calling a Function

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

```
fn
```

10.3 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"
```

10.4 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"
```

10.4.1 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
```

10.5 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

11.1 Undefining (unsetting) a command line function

To delete a function definition from the environment, we use

```
unset -f function_name
```

11.2 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
```

13.1 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.

14.1 Without Quotes

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

14.2 With Quotes

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

14.2.1 "$@"

- "$@" 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.

14.2.2 "$*"

- "$*" 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

- **${variable#pattern}**: 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
```

- **${variable##pattern}**: 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
```

16.1 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.

17.1 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.

17.1.1 Syntax

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

17.1.2 Example

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

17.2 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.

17.2.1 Syntax

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

17.2.2 Example

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

17.2.3 Example

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

[client]
user = admin
EOF

cat config.txt
```

Looping through characters in a string

18.1 Read while Here string

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

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

18.2 Using grep

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

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

18.3 C-style for loop

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

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

18.4 Sequence loop

```
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.

19.1 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
```

19.2 Exponentiation

We use the carrot (∧) for exponentiation

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

19.3 Floating-Point Arithmetic

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

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

19.4 Converting Decimal to Binary

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

num=32

echo "obase=2; $num" | bc
```

19.5 Advanced floating-point arithmetic

In general when we are dealing with floating point arithmetic in scripts, we should pipe the result into bc -l.

For example:

```
local x y sum_squares res
x=$1;y=$2
res=0

sum_squares=$(echo "$x^2 + $y^2" | bc -l)

if (( $(echo "$sum_squares <= 1" | bc -l)  )); then
    (( res+=10 ))
elif (( $(echo "$sum_squares <= 25"  | bc -l) )); then
    (( res+= 5 ))
elif (( $(echo "$sum_squares <= 100" | bc -l) )); then
    (( res+= 1 ))
fi

echo $res
```

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.

20.1 Syntax

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

```

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

20.2 Format Specifiers

- **Integer Format Specifiers**
  - `%d` or `%i`: Signed decimal integer.
  - `%u`: Unsigned decimal integer.
  - `%o`: Unsigned octal integer.
  - `%x`: Unsigned hexadecimal integer (lowercase letters).
  - `%X`: Unsigned hexadecimal integer (uppercase letters).
- **Floating-Point Format Specifiers**
  - `%f`: Floating-point number (fixed-point notation).
  - `%F`: Floating-point number (fixed-point notation, uppercase).
  - `%e`: Floating-point number (scientific notation, lowercase).
  - `%E`: Floating-point number (scientific notation, uppercase).
  - `%g`: Floating-point number (uses `%e` or `%f` format, whichever is shorter).
  - `%G`: Floating-point number (uses `%E` or `%F` format, whichever is shorter).
- **Character and String Format Specifiers**
  - `%c`: Single character.
  - `%s`: String.
- **Pointer Format Specifier**
  - `%p`: Pointer address.

- **Special Characters**
  - `%%`: A literal `%` character.
- **Width and Precision**
  - `%N.d`: Width `N`, with `d` decimal places for floating-point numbers (e.g., `%6.2f`).
  - `%-N`: Left-align within the specified width `N`.
- **Example Usages**
  - **Integer with leading zeros**: `printf "%05d\n" 42` outputs `00042`.
  - **Floating-point with precision**: `printf "%.2f\n" 3.14159` outputs `3.14`.
  - **Left-aligned string**: `printf "%-10s\n" "bash"` outputs `bash      `.
- **Flags**
  - `-`: Left-align the output within the specified field width.
  - `+`: Force a sign (`+` or `-`) to be used on a number.
  - `0`: Pad the field with leading zeros.
  - `#`: Use an alternate form: prefix for octal `0`, prefix for hexadecimal `0x` or `0X`.
- **Examples with Flags**
  - **Left-aligned integer**: `printf "%-5d\n" 42` outputs `42   `.
  - **Force sign**: `printf "%+d\n" 42` outputs `+42`.
  - **Alternate form**: `printf "%#x\n" 42` outputs `0x2a`.

More on variable assignment

21.1 Let

The let command in Bash is used to perform arithmetic operations. It allows for calculations to be done directly within a script. Here is an explanation of let along with some examples of how it can be used:

21.2 Syntax

```
let expression
```

21.3 Features

- **No Need for $(()):** With let, you don't need to use $(()) for arithmetic expressions.
- **Multiple Expressions:** You can evaluate multiple expressions separated by spaces.
- **Return Status:** let returns 1 if the expression evaluates to 0, and 0 if the expression evaluates to a non-zero value.

21.4 Basic Usage

```
count=0

let count=count+1
let count+=1
let count++
let ++count
```

21.5 readonly

Marks a variable as read-only, preventing any modification after its initial assignment.

```
readonly var=value

readonly a=15
```

21.6 declare and typeset

Used to declare variables and set their attributes. The typeset command is often considered synonymous with declare in many shells.

```
declare VAR_NAME=value
declare -r VAR_NAME=value   # Read-only
declare -i VAR_NAME=10      # Integer
declare -a VAR_NAME         # Array
declare -A VAR_NAME         # Associative array

typeset VAR_NAME=value
typeset -r VAR_NAME=value   # Read-only
typeset -i VAR_NAME=10      # Integer
typeset -a VAR_NAME         # Array
typeset -A VAR_NAME         # Associative array
```

Understand the ”for item in list” loop

22.1 What we mean by ”list”

To understand this type of loop we first need to understand what we mean when we say ”lists”.

The term list can mean array, but it can also mean something like a space separated string.

22.1.1 Strings separated by delimiters defined in the IFS

This is a simpler form of list but lacks the advanced features of arrays.

```
fruits="apple banana cherry"

# Iterating over space-separated string
for fruit in $fruits; do
    echo "Fruit: $fruit"
done

IFS=:
fruits="apple:banana:cherry"

for fruit in $fruits; do
    echo "$fruit"
done
```

22.2 The loop

The for item in loop in Bash is a versatile and commonly used loop structure that iterates over a list of items. This loop allows you to execute a set of commands for each item in the list.

hen you provide a space-separated list to a for loop, Bash splits the list into individual words based on these delimiters.

```
list="apple banana cherry"
for item in $list; do
    echo "Item: $item"
done
```

Note:-

Its crucial that $list remains unquoted, this allows bash to perform word splitting based on the IFS variable.

for i; Looping through each argument

the for i; do ... done syntax in Bash is a shorthand for looping through each positional parameter (argument) passed to the script. This is a simplified way of writing for i in "$@"; do ... done, where "$@" represents all the positional parameters.

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

for i; do
    echo $i
done

// Calling with ./script "hello" "world"
// Output:
hello
world
```

Converting from string to int: Arithmetic expansion base prefixs

In arithmetic expansion, you can specify the base (radix) of the number using a prefix before the number. Common prefixes include:

- **2#**: for binary (base-2)
- **8#**: for octal (base-8)
- **10#**: for decimal (base-10)
- **16#**: for hexadecimal (base-16)

Effect of 10# Prefix: By using 10#, you explicitly tell Bash to interpret the following string as a base-10 number, which effectively removes any leading zeros because the leading zeros are irrelevant in base-10 arithmetic.

```
str="0010"
stoi=$((10#$str)) && echo $stoi

// Or simply

echo "$((10#$str))"
```

Associative array (maps)

In Bash, associative arrays must be explicitly declared using declare -A (or typeset -A in some shells) before you can use them. This is because Bash needs to know that you intend to use string keys for your array, which is a different behavior from standard indexed arrays.

```
declare -A arr

arr["key1"]=val1
arr["key2"]=val2

echo "${arr["key1"]}"
```

```
  declare -A abilities=(
      [strength]=''
      [dexterity]=''
      [constitution]=''
      [intelligence]=''
      [wisdom]=''
      [charisma]=''
)
```

25.1 Looping through the map

In Bash, when you loop through an associative array using a for loop, you need the ! symbol to get the list of keys from the array.

```
 declare -A arr

arr["key1"]=val1
arr["key2"]=val2

for key in "${!arr[@]}"; do
    echo "Key: ${key}  Value: ${arr[$key]}"
done

// Output:
Key: key2  Value: 2
Key: key3  Value: 3
Key: key1  Value: 1
```

Note:-

The order in which keys are iterated in an associative array in Bash is not guaranteed. Associative arrays in Bash do not maintain insertion order

Random

Bash provides several methods to generate random numbers, including the use of built-in variables, external commands, and utilities.

26.1 $RANDOM

Bash has a built-in variable called $RANDOM that generates a pseudo-random integer in the range 0 to 32767.

```
random_number=$RANDOM
echo "Random number: $random_number"
```

26.1.1 Generating Random Numbers in a Range

You can generate random numbers within a specific range by using modulo arithmetic and adjusting the base.

```
# Generate a random number between 1 and 100
random_number=$(( (RANDOM % 100) + 1 ))
echo "Random number between 1 and 100: $random_number
```

26.2 Using shuf

The shuf command can be used to generate random numbers or shuffle lines in a file. It is more flexible than *RANDOM*.

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

# Generate a random number between 1 and 100 using shuf
random_number=$(shuf -i 1-100 -n 1)
echo "Random number between 1 and 100: $random_number"
```

26.3 Using /dev/urandom or /dev/random

```
random_number=$(od -An -N1 -i /dev/urandom)
```

26.3.1 Breakdown

- **/dev/urandom:**
  - /dev/urandom is a special file in Unix-like operating systems that provides random data. It is suitable for most cryptographic purposes and is non-blocking, meaning it won't wait for high-quality entropy and will return data as soon as it is available.
- **od Command:**

  od stands for "octal dump," but it can also be used to display data in various formats, including decimal, hexadecimal, and ASCII. It reads binary data from a file (or standard input) and formats it for output.
- **Options Used with od:**
  - **-A n:** Suppresses the output of file offsets (addresses). Normally, od outputs the byte offsets at the beginning of each line. -A n tells od not to print these offsets.
  - **-N1:** Tells od to read only one byte of input.
  - **-i:** Interprets the input as a signed decimal integer

26.4 Random Floating-Point Number with awk

Bash does not support floating-point arithmetic directly, but you can use awk or bc to generate random floating-point numbers.

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

# Generate a random floating-point number between 0 and 1 using
↪  awk
random_float=$(awk -v seed=$RANDOM 'BEGIN { srand(seed); print
↪  rand() }')
echo "Random floating-point number between 0 and 1:
↪  $random_float"
```

Shift

The shift keyword in Bash is used to manipulate the positional parameters of a script or function. Specifically, it shifts the positional parameters to the left by a specified number of positions. When you use shift, the value of $1 is discarded, $2 becomes $1, $3 becomes $2, and so on. This is useful for processing command-line arguments one at a time.

27.1 Syntax

```
shift [n]
```

- **n:** The number of positions to shift. If n is not specified, it defaults to 1.

unset

The unset command in Bash is used to remove variables or functions from the shell environment. When a variable or function is unset, it is no longer recognized by the shell and cannot be used until it is redefined.

28.1 Syntax

```
unset [OPTION] NAME...
```

- **NAME:** The name of the variable or function to be unset.
- **[OPTION]:** Optional flags to modify the behavior of unset.

28.2 Options

- **-v:** Treats the NAME arguments as variable names. This is the default behavior.
- **-f:** Treats the NAME arguments as function names

28.3 Unsetting Variables

```
myvar="Hello, World!"
echo $myvar  # Output: Hello, World!

unset myvar
echo $myvar  # Output: (empty, since myvar is unset)
```

28.4 Unsetting Array Elements

```
myarray=(apple banana cherry)
unset 'myarray[1]'  # Note the use of quotes
echo "${myarray[@]}"  # Output: apple cherry
```

28.5 Unsetting Functions

```
myfunc() {
    echo "This is a function"
}
myfunc  # Output: This is a function

unset -f myfunc
myfunc  # Output: (command not found, since myfunc is unset)
```

Processing command line options

29.1 Without getopts

```
while [[ -n "$(echo $1 | grep '-')" ]]; do
    case "$1" in
        -a) process -a ;;
        -b) process -b with arg as $2
            shift ;;
        -c) process -c ;;
        *) echo "Usage: ./script [-a] [-b barg] [-c args...]"
    esac
    shift
done
```

29.2 With getopts

getopts is a built-in command in Bash that simplifies parsing command-line options and arguments. It handles both short options (e.g., -a) and options with arguments (e.g., -b value).

29.2.1 Syntax

```
while getopts "options_string" opt; do
    case $opt in
        opt1) ;;
        opt2) ;;
        opt3) ;;

        ...

        *) ;;
    esac
done
```

29.2.2 Options string

Suppose we wanted our script to be run with three optional options a, b, and c. Our options string would be

```
"abc"
```

If we wanted these options to take in arguments, we add a colon after the option name

```
"a:b:c:"
```

29.2.3 Getting the options arguments (OPTARG)

OPTARG is a special variable used in conjunction with the getopts built-in command in Bash to retrieve the argument value associated with an option. When you specify that an option requires an argument (by appending a colon : to the option character in the options string), getopts automatically assigns the argument to OPTARG.

```
while getopts "ab:c:" opt; do
  case $opt in
    a)
      echo "Option -a triggered"
      ;;
    b)
      echo "Option -b triggered with argument: $OPTARG"
      ;;
    c)
      echo "Option -c triggered with argument: $OPTARG"
      ;;
    *)
      usage
      ;;
  esac
done
```

29.2.4 Shifting after processing with getopts

The command shift $((OPTIND - 1)) is used in a Bash script to adjust the positional parameters ($1, $2, etc.) after parsing options with getopts. This ensures that any remaining arguments (those not processed as options) are correctly indexed for further processing.

- getopts processes options passed to a script. As it processes each option, it increments a special variable called OPTIND (Option Index).
- OPTIND starts at 1 and points to the next argument to be processed. After getopts has processed all options, OPTIND holds the index of the first non-option argument.
- By using shift $((OPTIND - 1)), you adjust the positional parameters to remove the options processed by getopts. This means that after the shift, $1 will point to the first non-option argument.

A list of ALL IO redirectors

Redirector	Function
cmd1 | cmd2	Pipe; take standard output of cmd1 as standard input to cmd2.
> file	Direct standard output to file.
< file	Take standard input from file.
>> file	Direct standard output to file; append to file if it already exists.
>|file	Force standard output to file even if noclobber is set.
n>|file	Force output to file from file descriptor n even if noclobber is set.
<> file	Use file as both standard input and standard output.
n<> file	Use file as both input and output for file descriptor n.
<< label	Here-document; see text.
n > file	Direct file descriptor n to file.
n < file	Take file descriptor n from file.
n >> file	Direct file descriptor n to file; append to file if it already exists.
n>&	Duplicate standard output to file descriptor n.
n<&	Duplicate standard input from file descriptor n.
n>&m	File descriptor n is made to be a copy of the output file descriptor.
n<&m	File descriptor n is made to be a copy of the input file descriptor.
&>file	Directs standard output and standard error to file.
&<-	Close the standard input.
&>-	Close the standard output.
n>&-	Close the output from file descriptor n.
n<&-	Close the input from file descriptor n.
n>&word	If n is not specified, the standard output (file descriptor 1) is used. If the digits in word do not specify a file descriptor open for output, a redirection error occurs. As a special case, if n is omitted, and word does not expand to one or more digits, the standard output and standard error are redirected as described previously.
n<&word	If word expands to one or more digits, the file descriptor denoted by n is made to be a copy of that file descriptor. If the digits in word do not specify a file descriptor open for input, a redirection error occurs. If word evaluates to -, file descriptor n is closed. If n is not specified, the standard input (file descriptor 0) is used.
n>&digit-	Moves the file descriptor digit to file descriptor n, or the standard output (file descriptor 1) if n is not specified.
n<&digit-	Moves the file descriptor digit to file descriptor n, or the standard input (file descriptor 0) if n is not specified. digit is closed after being duplicated to n.

Exercise solutions

31.1 Exercise 1

31.1.1 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.

31.1.2 Solution 1

```
#!/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 $counter
}
main "${@}"
```

31.1.3 Alternative solution

```
#!/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' "$count"
}
% main "$@"
```

31.2 Exercise 2

31.2.1 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

31.2.2 Solution

```
#!/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 "${@}"
```

31.2.3 Alternative solution

```
#!/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 "$@"
```

31.3 Exercise 3

31.3.1 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:

31.3.2 Solution

```
#!/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 "$count"
}
main "$@"
```

31.3.3 Alternate solution

```
#!/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
```

31.4 Exercise 4

31.4.1 Problem Statement

Create an implementation of the atbash cipher, an ancient encryption system created in the Middle East.

The Atbash cipher is a simple substitution cipher that relies on transposing all the letters in the alphabet such that the resulting alphabet is backwards. The first letter is replaced with the last letter, the second with the second-last, and so on.

An Atbash cipher for the Latin alphabet would be as follows:

Plain: abcdefghijklmnopqrstuvwxyz
Cipher: zyxwvutsrqponmlkjihgfedcba

It is a very weak cipher because it only has one possible key, and it is a simple mono-alphabetic substitution cipher. However, this may not have been an issue in the cipher's time.

Ciphertext is written out in groups of fixed length, the traditional group size being 5 letters, leaving numbers unchanged, and punctuation is excluded. This is to make it harder to guess things based on word boundaries. All text will be encoded as lowercase letters.

31.4.2 Solution

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

main() {

    result=$( tr "abcdefghijklmnopqrstuvwxyz"
    ↪  "zyxwvutsrqponmlkjihgfedcba" <<< ${2,,} | tr -d " ," )

    [ "$1" == "encode" ] && result=$(fold -w 5 <<< $result)

    echo $result
}
main "$@"
```

31.5 Exercise 5

31.5.1 Problem Statement

Calculate the points scored in a single toss of a Darts game.

Darts is a game where players throw darts at a target.

In our particular instance of the game, the target rewards 4 different amounts of points, depending on where the dart lands:

- If the dart lands outside the target, player earns no points (0 points).
- If the dart lands in the outer circle of the target, player earns 1 point.
- If the dart lands in the middle circle of the target, player earns 5 points.
- If the dart lands in the inner circle of the target, player earns 10 points.

The outer circle has a radius of 10 units (this is equivalent to the total radius for the entire target), the middle circle a radius of 5 units, and the inner circle a radius of 1. Of course, they are all centered at the same point — that is, the circles are concentric defined by the coordinates (0, 0).

Given a point in the target (defined by its Cartesian coordinates x and y, where x and y are real), calculate the correct score earned by a dart landing at that point.

31.5.2 Basic Solution

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

function quit {
    echo "# there is _some_ output" && exit 1
}


function main {

    (( $# != 2 ))  && quit

    if ! [[ $1 =~ ^-?[0-9]+(\.[0-9]+)?$ && $2 =~
    ↪  ^-?[0-9]+(\.[0-9]+)?$ ]]; then
        quit
    fi

    local x y sum_squares res
    x=$1;y=$2
    res=0

    sum_squares=$(echo "$x^2 + $y^2" | bc -l)

    if (( $(echo "$sum_squares <= 1" | bc -l)  )); then
        (( res+=10 ))
    elif (( $(echo "$sum_squares <= 25"  | bc -l) )); then
        (( res+= 5 ))
    elif (( $(echo "$sum_squares <= 100" | bc -l) )); then
        (( res+= 1 ))
    fi

    echo $res
}
main "$@"
```

31.5.3 Advanced solution

```
#!/usr/bin/env bash
die () { echo "$1"; exit 1; }
(( $# != 2 )) && die "Invalid arg count"
for i; do [[ $i = *[^[:digit:].-]* ]] && die "Non-numeric arg"; done
bc <<< "scale=4
        x=$1 ; y=$2 ; d=sqrt(x^2 + y^2 )
        if (d <= 1) 10 else if (d <= 5) 5 else if (d <= 10) 1 else 0"
exit 0
```