

More on Shell

Petr Ospalý

Sep 2025

GNU (GNU's Not UNIX)



- Initially GNU supposed to be fully functional OS based on microkernel (hurd) but it never came to be and instead GNU got stuck with vast ecosystem of utilities but no usable kernel – that changed with **Linux**
- Many components distributed with Linux distribution come from the GNU project – most of them can be replaced with alternatives except **glibc** (std. C library) and **gcc** (C compiler) for which Linux kernel depends on
- GNU zealots will demand the use of the term **GNU/Linux** – feel free to ignore them :)
- Next to the **gcc** compiler the other most influential contribution to the world at large were **GPL** (copyleft) licenses – **Linus Torvalds** attributes the success of Linux to picking the **GPL 2.0** license for it

REPL

- **Read-Eval-Print-Loop**
- Used in interpreted languages and shell environments (like in python, smart switch CLI, UNIX shell, browser javascript console etc.)
- It is command line interface where instructions to the interpreter are separated by newline – interpreter **reads** the command (hitting ENTER key will **evaluate**/execute the command)
- *Long lines could usually be split visually (with backslash technique)*
- Each command then can **print** out some output (does not need to) and interface is ready for another input (**looping back** to read)

GNU BASH

- **Interactive shell** of our choice during this course will be **BASH** (Bourne-Again SHell)
- It builds upon the traditional UNIX shell:
Bourne Shell
- Scripts in this course will target **POSIX** shell specification avoiding any **bashisms**
- The reasons to avoid scripting for BASH:
 - Bash is not a default shell everywhere and installing it is not always option
 - BSD *NIX systems use csh, ksh and other shells
 - “Minishells” like **Busybox** are also commonly encountered and will not support bashisms
- The reasons to use BASH as login/interactive shell:
 - More pleasant to use with many conveniences (next slide)
 - Feel free to replace Bash with **zsh** (more features and more conveniences)

GNU Readline

- Bash is using **readline library** for its interactive input which provides many nice features:
 - **TAB completion** (killer feature for interactive shell) – hitting TAB key on the keyboard will complete the command or list possible options on double-tap[*]
 - Command history
 - Line editing
- **[*] Learn to use TAB completion – it will change the way you use your terminal and shell...**
- There are two modes to use the readline (“**set -o emacs**” vs “**set -o vi**”):
 - EMACS mode (the default) – this presentation will work with the default mode
 - VI mode (as a Vim user I suggest to try this mode for VI-enthusiast)
- *Next text will use terms like Bash, shell, command line etc. interchangeably without mentioning readline again*

Pagers

- Sometimes (often times) the program output has too many lines to fit in your terminal window or console screen – in that case chunk of it will scroll out of sight
- Terminals usually have scroll bars so there it is not as big issue but virtual console or serial console will not have them...
- The solution is to store the output in a file (more on that later elsewhere) or by using **pager**
- **Pager** is a tool which will enable “paging” the output in your terminal screen one page at a time so you can see the whole output
- Some pagers will not allow to return back by page (old “**more**” command)
- **Many in use will enable back and forth scrolling, searching and quick navigation**

History

- Bash has “**history**” command which will output all commands used in the past – up to a set limit in shell variable **HISTSIZE** (on variables soon)
- The history is stored in a file defined by shell variable **HISTFILE** – default value points to inside a user directory as a dot-file (“hidden” file): `.bash_history`
- To use the history hit arrow keys (up/down) in shell prompt to reuse commands from the past
- Bash also can “search” the history by repeated hitting **CTRL-r** (backward search) and typing the pattern to search for

Manpages (1/3)

- *Long overdue introduction to this crucial faculty...*
- UNIX had brilliant built in documentation – the so called **manpages** (manual pages)
- UNIX-like systems still use these and they should be installed on any sensible Linux distribution – you can also find them [online](#)
- You can access manpage of any tool/command with the command: `man <command name>`
- Start with: **man man** (quit by hitting “q” key)

Manpages (2/3): man man

```
MAN(1)                                Manual pager utils                                MAN(1)
NAME
    man - an interface to the system reference manuals

SYNOPSIS
    man [man options] [[section] page ...] ...
    man -k [apropos options] regexp ...
    man -K [man options] [section] term ...
    man -f [whatis options] page ...
    man -l [man options] file ...
    man -w|-W [man options] page ...

DESCRIPTION
    man is the system's manual pager. Each page argument given to man is normally the name of a program, utility or function. The manual
    page associated with each of these arguments is then found and displayed. A section, if provided, will direct man to look only in that
    section of the manual. The default action is to search in all of the available sections following a pre-defined order (see DEFAULTS),
    and to show only the first page found, even if page exists in several sections.

    The table below shows the section numbers of the manual followed by the types of pages they contain.

    1 Executable programs or shell commands
    2 System calls (functions provided by the kernel)
    3 Library calls (functions within program libraries)
    4 Special files (usually found in /dev)
    5 File formats and conventions, e.g. /etc/passwd
    6 Games
    7 Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7), man-pages(7)
    8 System administration commands (usually only for root)
    9 Kernel routines [Non standard]

    A manual page consists of several sections.

    Conventional section names include NAME, SYNOPSIS, CONFIGURATION, DESCRIPTION, OPTIONS, EXIT STATUS, RETURN VALUE, ERRORS, ENVIRONMENT,
    FILES, VERSIONS, CONFORMING TO, NOTES, BUGS, EXAMPLE, AUTHORS, and SEE ALSO.

Manual page man(1) line 1 (press h for help or q to quit)
```

Manpages (3/3)

- Some names conflict; for example shell command **printf** and C function from standard library also named printf
- Typing “**man printf**” will return manpage of shell command which you will want most in this course but to get manpage for C function you must switch to different section: **man 3 printf**
- Opening manpage will bring you to a pager:
 - Arrow keys, page-down/up will enable navigation
 - Hitting “/” will bring up pattern search (bottom of screen)
 - Type search term and hit ENTER
 - Hitting lower-case “n” and upper-case “N” will cycle through found matches
 - Quit by hitting “q” key

Other help (1/2)

- Since we introduced manpages – it is good from now on to explore the documentation for every new command or tool we will mention... “**man <new tool>**”
- Apart from manpages other resources can be utilized
- There might be GNU “**info**” command for extended documentation of GNU programs (try “**info info**”)
- Sometimes you want to do something but you have no idea what utility provides the feature – in such case “**apropos**” command might be of use; the simplest use: `apropos <search term>`

It will search through manpages for this term, e.g. “**apropos editor**” will list every installed program which might mentioned the word “editor” in their name or description

- Many commands have built-in help accessible via command argument: **-h** or **--help**

Other help (2/2)

```
% apropos --help
Usage: apropos [OPTION...] KEYWORD...
```

| | |
|-------------------------------------|--|
| -d, --debug | emit debugging messages |
| -v, --verbose | print verbose warning messages |
| -e, --exact | search each keyword for exact match |
| -r, --regex | interpret each keyword as a regex |
| -w, --wildcard | the keyword(s) contain wildcards |
| -a, --and | require all keywords to match |
| -l, --long | do not trim output to terminal width |
| -C, --config-file=FILE | use this user configuration file |
| -L, --locale=LOCALE | define the locale for this search |
| -m, --systems=SYSTEM | use manual pages from other systems |
| -M, --manpath=PATH | set search path for manual pages to PATH |
| -s, --sections=LIST, --section=LIST | search only these sections (colon-separated) |
| -, --help | give this help list |
| -, --usage | give a short usage message |
| -V, --version | print program version |

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

The --regex option is enabled by default.

Report bugs to cjwatson@debian.org.

Shell Variable (1/4)

- Shell/bash is not just an executor of commands (although that is its main purpose) – it is also a programming/scripting language
- *This series of presentation is not about shell scripting though...*
- Some programming constructs will be shown where appropriate
- **Nevertheless** – usage of **shell variables** even outside of shell script is common
- Variable name can be any **alphanumeric string including underscores**, e.g.: “_var”, “var1”, “MYVAR”, etc.
- Value can be any arbitrary string (even empty)
- In some contexts a string of digits can be interpreted as a numeric value (integer)
- Usually in scripts an **empty string serves as a boolean False**

Shell Variable (2/4)

- Syntax to set and declare a variable: `NAME=<value>`
(no space around the equal sign „=“)
- If the value contains „whitespace“ (spaces, newlines, tabs) then it must be either prefixed with backslash or enclosed in single or double-quotes:
`value\ with\ spaces\ in\ it`
`'value with spaces in it'`
`"value with spaces in it"`
- The shell variable is then used via **variable expansion** by prefixing the name with a dollar sign “\$”:
`$myvar` → will return the value
- This syntax cannot be recommended though due to the way shell parse and evaluate these expansions... (see on the next slide)
- If you want to “delete” the variable then it must be **unset**: `unset myvar`

Shell Variable (3/4)

- Always wrap variable name in curly braces and surround it with double-quotes: **"\${myvar}"** → **the correct way** (you cannot make a mistake with it ever...)
- Best way to showcase the differences is with examples, let's have this definition:
`var='123'`
(added quotes but not needed in this particular case – **defensive programming**)
- Let's use the variable in the following expansions:
 - `$var` → 123
 - `${var}` → 123
 - `'${var}'` → `${var}` → *single quotes prevent expansion*
 - `\${var}` → `${var}` → *same with backslash*
 - `"${var}"` → 123 → *this is the correct way*
 - `$variable` → (EMPTY) → *we turned "var" to "variable" which is **undefined***
 - `${var}iable` → 123iable → *curly braces fixed it*
 - `"${var}iable"` → 123iable → *the correct way*

Shell Variable (4/4)

- Syntax recap:
 - Dollar sign „\$“ triggers expansion
 - Curly braces separates variable name from surrounding text
 - Single quotes prevent any expansion
 - Backslash in front of dollar sign prevent the expansion too „\\$“
 - Double quotes enables expansion and hold string with spaces together as one token
- *The reason why quotes are needed will be apparent in more advanced examples elsewhere*
- The „\\$“ sequence is called **escape sequence**... backslash is used this way to halt any special meaning a symbol can have... for example:
 - ‘this string\'s value is inside single quotes’
 - “this string shows one backslash (\\) and one double-quote (\\”)”

Shell Environment (1/4)

- Shell will create an **environment** (set of variables) upon it's start
- These variables can be changed any time during the session or prior inside a shell initialization files
- To see the current status of environment use this command: **printenv**
- List of environment variables will differ based on OS, context, desktop, global settings, user setup etc.
- User defined shell variable (e.g. MYVAR) can be „exported“ to the environment as: **export MYVAR**
- There is no restriction on variable names but one can choose lower case letters for variable names to avoid conflicts with implicit and default environment variables which are canonically upper case
- Prefixing/postfixing with an underscore also works: **_MYVAR**

Shell Environment (2/4)

- Below are variables good to know and recognize:
- **IFS**: Actually content of this variable will decide how string is split into tokens/fields and although the default **is** whitespaces but it does not need to be: <space><tab><newline>
- **PWD**: Value is the same as of the **pwd** command
- **HOME**: Current user's home directory; "tilde" character "~" can be used instead – it will be expanded by the shell to actual filepath
- **USER**: Current username
- **SHELL**: Current running shell
- **TERM**: Virtual terminal emulation mode (e.g. xterm)
- **EDITOR**: This will set the default editor program for shell and many utilities (like visudo, vipw, git, etc.) - on editors later
- **DISPLAY**: Set for **X.org** desktop (it tells the GUI app where to draw a window; useful to modify in case there are multiple **xservers** available)

Shell Environment (3/4)

- **PATH:** Contains filepaths/directories separated by colons ":" - e.g.:
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
 - PATH variable is used when user try to run a command and the command name is not an absolute nor relative path
 - Shell will try to locate the **executable** of the same name as the command within these directories and in this order
 - This command will ***not*** use PATH variable: **./myscript**
 - This command ***will use*** PATH variable: **myscript**

It is often needed to modify the PATH so your custom locations can be searched too – for example we might want to add our own “~/bin” directory:

`PATH="${PATH}:${HOME}/bin"` → notice the expansions

Shell Environment (4/4)

- **LANG, LC_ALL, LC_***: Determines locale for the session and language in which commands should return their output if they support the locale – the basic value “**C**” should be always implemented (English)
- **LC_ALL** has the highest precedence and overwrites every category **LC_*** and **LANG**
- **LANG** is used if **LC_ALL** is not set and when some **LC_*** category is unset
- **LANG=C** or **LC_ALL=C** is useful fix if the system has set foreign incomprehensible locale
- To see what locales are supported in the system:
`locale -a`
- To get current locales:
`locale`

```
% locale
LANG=cs_CZ.UTF-8
LC_CTYPE="cs_CZ.UTF-8"
LC_NUMERIC="cs_CZ.UTF-8"
LC_TIME="cs_CZ.UTF-8"
LC_COLLATE="cs_CZ.UTF-8"
LC_MONETARY="cs_CZ.UTF-8"
LC_MESSAGES="cs_CZ.UTF-8"
LC_PAPER="cs_CZ.UTF-8"
LC_NAME="cs_CZ.UTF-8"
LC_ADDRESS="cs_CZ.UTF-8"
LC_TELEPHONE="cs_CZ.UTF-8"
LC_MEASUREMENT="cs_CZ.UTF-8"
LC_IDENTIFICATION="cs_CZ.UTF-8"
LC_ALL=
```


Globbs: * (1/3)

- We just mentioned shell expansion of a variable: \$VAR; which is basically a **placeholder** for its value
- Very common feature of a shell is pattern-matching which uses special characters to match filenames (**globbs**)
- **Do not confuse with regular expressions – those are much more powerful and way more complicated**
- The glob is then “**expanded**” to its matched filenames similarly as shell variable expansion is expanded to its value
- Globbs are very simple and limited – think of them as wildcards; E.g.: *, ***hoj**, **a***, **a*j** will all match the word: **ahoj**

Globbs: * (2/3)

- Asterisk will match everything – except „hidden“ (dot) files – to match those a separate „.*“ must be used
- The simplest kind of glob expression is asterisk surrounded by characters from one or both sides
- **If there is no file then glob expression will be used literally...** (in this case a single asterisk)

```
/tmp/empty % ls -la
```

```
.  
..
```

```
/tmp/empty % ls *
```

```
ls: cannot access '*': No such file or directory
```

```
/tmp/empty %
```

```
/tmp/globs % ls -la
```

```
.  
..
```

```
.abc
```

```
ALPHA
```

```
Abc
```

```
abc
```

```
alphabet
```

```
beta
```

```
/tmp/globs % ls -1 *
```

```
ALPHA
```

```
Abc
```

```
abc
```

```
alphabet
```

```
beta
```

```
/tmp/globs % ls -1 .*
```

```
.abc
```

```
/tmp/globs %
```

Globbs: * (3/3)

- Extra examples of globbs (for curious)
- ? matches one character exactly, [] defines a set of characters to match, [!] defines exclusionary set (*both support range like 0-9 and a-z*)
- Notice that asterisk is hungry
- For any advance matching logic we have to use **regexp** instead... (e.g. list only names without “e” – no need to understand this):

```
/tmp/globs % ls -1 | grep '^[^e]\+$'
ALPHA
Abc
abc
/tmp/globs %
```

```
/tmp/globs % ls -1
```

```
ALPHA
```

```
Abc
```

```
abc
```

```
alphabet
```

```
beta
```

```
/tmp/globs % ls ???
```

```
Abc abc
```

```
/tmp/globs % ls [Aa]*
```

```
ALPHA Abc abc alphabet
```

```
/tmp/globs % ls [!ab]*
```

```
ALPHA Abc
```

```
/tmp/globs % ls *[^e]* # See?
```

```
ALPHA Abc abc alphabet beta
```

```
/tmp/globs % ls *[^e]? # My point
```

```
ALPHA Abc abc beta
```

```
/tmp/globs %
```

Globs are limited – you will get the most mileage out of the single asterisk “*” alone.

Shell builtins (1/2)

- So far we used terms like command and programs interchangeably but there is difference:
 - program or binary is separate executable somewhere in the filesystem
 - script is separate text file interpreted as executable
 - shell **builtin** is integrated part of the shell itself and does not have separate binary somewhere in the FS
- In most cases we don't need to care unless we have a binary which we want to use but it is „masked“ by a builtin
- Or we have more than one version of a program in separate locations and we are not sure which is actually used...

Shell builtins (2/2)

- To decipher any command we can use the command (☺)

„**type**“: `type <command>`

- Let's try it on **cd**: `type cd`
`cd is a shell builtin`

```
/ % type type
type is a shell builtin
/ %
```

- So every time we use **cd** we are using shell builtin function
- If we try to read its documentation (**man cd**) then we will get full manual page for bash itself – all bash features including builtins are described there but...
- ...there is a better way to get help with builtins: **help cd**

Comments & echo

- Builtin **echo** is the traditional „printing“ command in shell
- For scripting purposes though it might be better to replace it with **printf** (explained elsewhere)
- Comment in shell starts with „#“ and continues until the end of the line – everything between hash character and the newline is ignored
- Picture is worth of thousands words...

```
/ % type echo
echo is a shell builtin
/ %
/ % myvar='This is my var...'
/ % echo "${myvar}"
This is my var...
/ % echo "${myvar}printed by echo"
This is my var...printed by echo
/ % echo "${myvar}printed by echo" "3 times..."
This is my var...printed by echo 3 times...
/ %
/ % # Hash character (#) starts comment...
/ % echo "This is printed..." # but this will not
This is printed...
/ %
```



Executables (1/5)

- Let's recap what executable on a *NIX system is:
 - It can be a binary (program)
 - It can also be a text file aka a script if it starts (on very first line) with so called **shebang** (**#!**) **line**:
#!<path to the interpreter>
 - Both binary and script must have execution bit set to be executable (*adding „exe“ extension like in DOS/Win does not make file executable*)
- Executables are searched in env. variable **PATH**, e.g.:
`PATH='/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin'`

Executables (2/5)

- When executable is invoked without a path (absolute nor relative) then executable must be located in one of these locations in **PATH** (except shell builtins for obvious reasons)
- If the executable is not in any of the locations defined in **PATH** then it must be invoked with explicit location, e.g.: `~/myapps/someapp.bin`
- **Note:** If your **PWD** (current directory) is the same as the executable then you still must prefix the name with „./“ specifying current directory (unless you add „./“ into the PATH – **don't do that...**)

***Why?** Imagine: bad actor leaves a malicious binary in often used directory – e.g. **passwd**; what happens if user tries to change their password? Danger is mitigated by explicitly adding „./“ to run command from current directory.*

Executables (3/5)

- **PATH** offers multiple locations for executables/commands
- To discover conflicting commands use: “**whereis**” and “**which**”
- **whereis**: returns all locations of the command name and manpages if they exist
- **which**: will return one executable out of many potential options (as shown by *whereis*)
 - One out of all executables “which” will be used

```
~ % whereis ls
ls: /usr/bin/ls /usr/share/man/man1/ls.1.gz
~ % which ls
alias ls='ls --color=auto'
      /usr/bin/ls
~ % /usr/bin/ls -1 /usr/local/
bin
etc
games
include
lib
lib64
libexec
sbin
share
src
~ %
```

In this case „**which**“ returns an **alias** and the aliased binary (aliases will be covered soon...)

Executables (4/5)

- Curious case of **cd**...
- **whereis** and **which** will report command location at **/usr/bin/cd** but when we try to use that program – it seems to not work...
- ...on the other hand the builtin works correctly
- Let's run command **"file"** for further analysis – it returns filetype description
- The reason is: **/usr/bin/cd** is actually a script which calls **"builtin cd"** inside but in a **subshell** and that does not affect the current shell and the current location...

```
/ % pwd
/  
/ % whereis cd
cd: /usr/bin/cd /usr/share/man/man1/cd.1.gz  
/ % which cd
/usr/bin/cd  
/ % type cd
cd is a shell builtin  
/ % /usr/bin/cd /usr/local/bin/  
/ % pwd
/  
/ % cd /usr/local/bin/  
/usr/local/bin % pwd
/usr/local/bin  
/usr/local/bin % file /usr/bin/cd
/usr/bin/cd: a /usr/bin/sh script, ASCII text executable  
/usr/local/bin %
```

„file“ command will attempt to detect what kind of a file the argument is based on its content.

Why such script even exist? Just to satisfy **POSIX** standard...

```
#!/usr/bin/sh
builtin cd "$@"
```

Executables (5/5)

```
/ % cp -a /usr/bin/cd /usr/local/bin/  
/ % whereis -b cd  
cd: /usr/bin/cd /usr/local/bin/cd  
/ % which cd  
/usr/local/bin/cd  
/ % echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin  
/ %
```

- Let's duplicate the **cd** script to a new path
- This time **whereis** returns two executable options (option “-b” will print only binaries and skips manpages)
- And **which** returns first match based on the order of directories in the **PATH** (We skipped quotes around \$PATH here as the value is “sane” but we should follow our own advice and use quotes every time – one day it might bite us...)

Shell aliases (1/4)

- We encountered an **alias** multiple times already with the command **ls**

```
/ % type ls
ls is aliased to `ls --color=auto'
/ %
/ % type alias
alias is a shell builtin
/ %
```


Shell aliases (2/4)

- Running the builtin **alias** as is will print out all currently configured aliases...
- The output is actually identical to the way these aliases could be set – meaning; alias **ls** is setup with a command:
`alias ls='ls --color=auto'`
- **So what is alias?** It is simply a **shorthand** for the value – every time you use the alias it is like typing it's value instead

```
/ % alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias mc='. /usr/libexec/mc/mc-wrapper.sh'
alias xzegrep='xzegrep --color=auto'
alias xzfgrep='xzfgrep --color=auto'
alias xzgrep='xzgrep --color=auto'
alias zegrep='zegrep --color=auto'
alias zfgrep='zfgrep --color=auto'
alias zgrep='zgrep --color=auto'
/ %
```

Shell aliases (3/4)

- We can always add more aliases to the current shell session; e.g. alias below ("*hide*") creates a hidden directory (leading dot) and moves all visible files in the current directory to it:

```
alias hide='mv -b * .hidden/'
```

This is a silly command just to demonstrate that even shell expansion and globs (the asterisk „“) works properly...*

- Also **TAB** will work to autocomplete for you so you can condense long commands to few clever aliases and pick the right one with few keystrokes; e.g. by typing out "go2" and hitting TAB:

```
alias go2music='cd "${HOME}/Music/'  
alias go2work='cd "${HOME}/Projects/'  
alias go2mydrive='cd "/mnt/bigdrive/'
```

Shell aliases (4/4)

- Aliases are useful to condense long list of arguments to a shortcut like here:

```
alias rbak='rsync -avzPHAX --numeric-ids '
```

- But also to create a “command” for often used **one-liner** or script which is too short to bother to store in some file and location:

```
alias jpg2pdf='for i in *.jpg ; do convert "$i"  
"${i%.jpg}.pdf" ; done'
```


Shell profiles (1/7)

- We just learned about the aliases and shell builtin **alias**
- But all our painfully setup aliases will disappear once we open another shell session... or close the current one...
- For that and other reasons every shell has a **profile file** – where we can preset and preserve our setup and do additional tasks like loading extra config files or run ad-hoc scripts

Shell profiles (2/7)

- When an **interactive shell** (that is not a login shell – e.g. **every new terminal window...**) is started, bash reads and executes commands from **~/.bashrc** (if that file exists).
- We can save our alias commands there and they will be restored with every new shell session
- We could also use a „*source*“ command
- Bash provides **source** builtin but the equivalent and traditional „*.*“ dot-source builtin is POSIX compliant

Shell profiles (3/7)

```
% help .
```

```
.: . filename [arguments]
```

Execute commands from a file in the current shell.

Read and execute commands from FILENAME in the current shell. The entries in \$PATH are used to find the directory containing FILENAME. If any ARGUMENTS are supplied, they become the positional parameters when FILENAME is executed.

Exit Status:

Returns the status of the last command executed in FILENAME; fails if FILENAME cannot be read.

Shell profiles (4/7)

- We could create a file like “*.mystuff.sh*” (any name will do and no need to be “hidden”) with content similar to this:

```
# my custom shell initialization file
alias rbak='rsync -avzPHAX --numeric-ids '
alias ls='ls --color=auto'
PATH="${PATH}:${HOME}/bin" # adjusting PATH
export PATH
```

- Then we simply run the following on the current shell session or add this line into „**.bashrc**“ (notice the dot):

```
. ~/.mystuff.sh
```

Shell profiles (5/7)

- When bash is invoked as an **interactive login shell** (or with the **--login** option), it will source **/etc/profile** (if it exists) and then it will read and execute the first file found in this order:
 - ~/.bash_profile
 - ~/.bash_login
 - ~/.profile → this file is used by other shells too – not just bash
- When an interactive login shell exits, or a non-interactive login shell executes the **exit** builtin command, then it executes **~/.bash_logout** and **/etc/bash.bash_logout** (if the files exist)

Shell profiles (6/7)

- Shell profile files are mainly for the **convenience of the user** and no script should be depending on some global environment setup...
- This course is concerned with using command line and so we care about conveniences in an **interactive shell** and therefore our configuration will be limited to **~/.bashrc**
- Typical things to set:
 - Aliases
 - Sourcing additional files and scripts
 - Environment variables like PATH and PS1

Shell profiles (7/7)

- It can be helpful to customize the shell prompt to show more info and/or use colors for better readability
- For that following shell variables can be used:
 - **PS1**: used as the primary prompt string
 - PS2: the secondary prompt string (default "> ")
 - There is also PS3 and PS4 but we will care only about PS1

To make your prompt little prettier:

```
PS1='[\u@\h] \A \w\n% '
```

\u → username

\h → hostname

\A → time

\w → current directory

\n → newline

You can colorized it too but the value would be indecipherable here.

There are generators online to try...

Unicode and UTF-8 (1/4)

- This topic would deserve a whole chapter but we touch upon it briefly
- Computers understand numbers (bits and bytes), „text“ is also stored as bits and bytes – so there must be a higher level logic to interpret those binary values
- This abstraction is achieved by „**character sets**“ and “**encodings**”
- **ASCII** is one of the oldest and ubiquitous character set – all characters in this set can be covered in one byte – it uses only 7 bits actually
- So ASCII is not only character set but also an “**encoding**” – the way character of a particular set is “encoded” – 7 bits need just one byte – one encoding cannot be simpler than that...
- Many “national” variations were created on top of it using the last bit which created a lot problems during the early internet era and file sharing because a document could be shown using wrong encoding (text would be mingled with “?” and similar)

Unicode and UTF-8 (2/4)

- The problem was solved by comprehensive and exhaustive international character set: **UNICODE**
- **Unicode** is another character set which associate a number to a character via “**code point**” and currently the standard defines over million symbols across all possible languages, including emojis
- Because Unicode character can be a big number which cannot be covered by a single byte – **multiple bytes** must be used to represent a single character
- Straightforward encoding can be UTF-32, UTF-16 which uses 4 bytes and 2 bytes for each character respectively – this of course can be wasteful for characters which has low numeric value like all ASCII characters (which values match exactly into Unicode set)

Unicode and UTF-8 (3/4)

- UTF-32 wastes a lot of space and UTF-16 cannot represent all Unicode characters
- **UTF-8** comes to rescue...it is a **variable-width encoding** (some character will need only one byte like ASCII and lesser used will need more bytes)
- It is the most widely used encoding on the internet and a default on Linux and other modern systems
- The takeaway here is: **Use Unicode and UTF-8**

Unicode and UTF-8 (4/4)

```
% locale
LANG=cs_CZ.UTF-8
LC_CTYPE="cs_CZ.UTF-8"
LC_NUMERIC="cs_CZ.UTF-8"
LC_TIME="cs_CZ.UTF-8"
LC_COLLATE="cs_CZ.UTF-8"
LC_MONETARY="cs_CZ.UTF-8"
LC_MESSAGES="cs_CZ.UTF-8"
LC_PAPER="cs_CZ.UTF-8"
LC_NAME="cs_CZ.UTF-8"
LC_ADDRESS="cs_CZ.UTF-8"
LC_TELEPHONE="cs_CZ.UTF-8"
LC_MEASUREMENT="cs_CZ.UTF-8"
LC_IDENTIFICATION="cs_CZ.UTF-8"
LC_ALL=
```

Czech

You can set your preferred language and encoding by setting up the locale environment variables as we shown earlier

```
$ locale
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

US English

Text Editor (1/2)

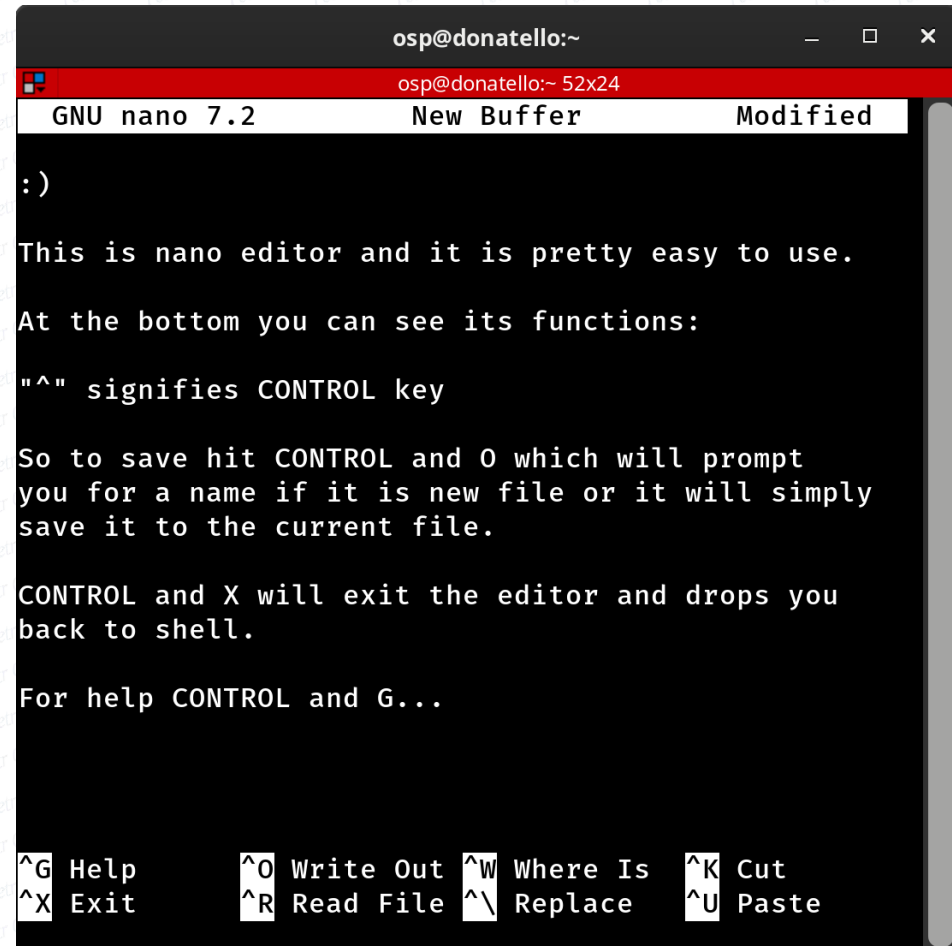
- It was mentioned before: **primary data type on a UNIX-like system is an ordinary text file**
- Config files, services, many system tools are just plain text (e.g. scripts, INI, JSON, YAML, XML etc.)
- **Text editor is then a major utility on a Linux server...**
- It is good to get familiar with at least one CLI editor because sometimes graphical text editor might not be available (on a remote server for example)
- **Vi** and improved **ViM** is the traditional UNIX editor but it might be too much for a novice (you might encounter “**emacs vs Vi**” flame-war – it is a running joke in the nix community)

Text Editor (2/2)

- **nano** might be a good alternative
- It is simple and intuitive
- Setup your **.bashrc** with:

```
EDITOR="nano"  
export EDITOR
```

It must be installed first of course...

A screenshot of the nano text editor running in a terminal window. The window title is 'osp@donatello:~'. The editor's status bar at the top shows 'GNU nano 7.2', 'New Buffer', and 'Modified'. The main editing area contains the following text: ':)', 'This is nano editor and it is pretty easy to use.', 'At the bottom you can see its functions:', '"^" signifies CONTROL key', 'So to save hit CONTROL and O which will prompt you for a name if it is new file or it will simply save it to the current file.', 'CONTROL and X will exit the editor and drops you back to shell.', and 'For help CONTROL and G...'. At the bottom of the editor, there is a help bar with the following shortcuts: '^G Help', '^O Write Out', '^W Where Is', '^K Cut', '^X Exit', '^R Read File', '^_ Replace', and '^U Paste'.

```
osp@donatello:~  
GNU nano 7.2      New Buffer      Modified  
:  
This is nano editor and it is pretty easy to use.  
At the bottom you can see its functions:  
"^" signifies CONTROL key  
So to save hit CONTROL and O which will prompt  
you for a name if it is new file or it will simply  
save it to the current file.  
CONTROL and X will exit the editor and drops you  
back to shell.  
For help CONTROL and G...  
  
^G Help      ^O Write Out  ^W Where Is  ^K Cut  
^X Exit      ^R Read File  ^_ Replace   ^U Paste
```

Final words

- **Learn to use manpages and use them often**
- Utilize TAB completion... save your fingers from typing...
- Leverage shell **history** – it is your journal and help
- Commands to remember when not sure:
 - **man** <command> → open manual page
 - **type** <command> → check what is the executable
 - **help** <builtin> → much easier than read through man page
 - **whereis/which** <binary/script> → to find out executable location
 - **file** <command> → to check type of the file
- Pick one and start using a CLI text editor (like **nano**)