210 Systems Programming
Writing Bash scripts

Fall 2025

Week 4

Overview

- Aliasing and *rc* (run commands) files
- Variables, environment variables
- Writing shell scripts
- Different types of quotes
- Command line arguments
- Bash programming constructs: conditionals, loops

Aliasing

`alias [-p] [name[=value]]`

- Use `alias` to save time by creating alternatives to commands with options
- '`alias`' with no arguments or with the -p option prints the list of aliases in the form alias `NAME=VALUE` on standard output.
- Otherwise, an alias is defined for each `NAME` whose `VALUE` is given.
- IMPORTANT: No spaces around =

Aliasing examples

- `alias mv='mv -i'`
  - Make prompting for overwrite during move a default option
- `alias ls='ls -al'`
  - Always list files in long format and show hidden files
- `alias ...='cd ../..'`
  - To rapidly move to levels up
- More examples: https://ostechnix.com/useful-bash-aliases/

Removing an alias

`unalias [-a] name [name ...]`

- Remove NAMEs from the list of defined aliases.
- If the -a option is given, then remove all alias definitions.

`alias` and `unalias` are built-in bash commands. So, use `help` instead of `man` to learn more about them.

How to make aliases permanent

- aliases are only active for the current bash session
- To re-activate them you need to re-write them at the beginning of every session
- We can write the aliases inside `.bashrc` to make them permanent
  - `.bashrc` is a "run-command", i.e., an `rc` file, which contains a list of bash commands to be executed at every bash launch
  - It is an *initialization* script

Variables

Shell Variables

NAME=[VALUE]

- Assign `VALUE` to the variable named `NAME`
- Accessible by the current shell session only. Can write them inside `.bashrc` to define them for every shell session.
- Providing no `VALUE` removes the previous assignment.
- Reference to *undefined* variables does not result in error. The result is just an empty string.

Variables

Environment Variables

- Accessible by the current shell session and **child processes**
- Use `export NAME=[VALUE]` to create them
- Required when created processes, such as `vim`, need access to the variables.
- Common environment variables: PATH, USER, SHELL, PWD
- Use `export` with no parameters to see the whole list

Variables

lvalues versus rvalues

- When setting the value of a variable (i.e., using the name as an lvalue), we use: `NAME=VALUE`
  - Example: `DATE=tuesday`
- When you want to retrieve the value of a variable and use it in an expression or in a command (i.e., rvalue), we use the $ prefix, as in `$NAME`
  - Example: `cp $FILE backup.txt`

Shell Scripts

shell script

A file containing a sequence of shell commands, functions, programming blocks. Used to perform complex shell operations.

- By convention, the file extension is `.sh`
  - Reminder: extension in Unix/Linux do not dictate file types
- Define the shell interpreter to use in the first line via:
  - `#!/bin/bash`
  - This is also called *shebang*
  - Needed for portability and POSIX compliance

Creating and executing shell scripts

- Can create via a text editor, such as vim
- You can also use echo statements to append lines to a script
- To execute a script within the current shell:
  - Use 'source script.sh' or '. script.sh'
  - This will execute the commands in the script as if they are entered within the current shell line by line
- To execute a script as a new process:
  - Make the script executable: `chmod +x script.sh`
  - Then run it as an executable program: `./script.sh`

Hello World! in Bash

```
#!/bin/sh
# This script prints Hello World!
echo Hello World!
# We could put more commands here
```

Another example

- Create the script: `vim myscript.sh`

```
#!/bin/sh
newdir=out
file=test.txt
mkdir -p $newdir
cd $newdir
echo "We generated this file" > $file
echo "and wrote to it" >> $file
cd ..
```

- Make it executable: `chmod +x myscript.sh`
- Run it: `./myscript.sh`

Quoting

- gnu.org reference manual on quoting
  - https://www.gnu.org/software/bash/manual/html_node/Quoting.html
- Three quoting mechanisms in Bash:
  - Escape character
  - Single quotes: `'text'`
  - Double quotes: `"text"`

The escape character

- It is considered as a quoting mechanism; because, it allows you to refer to a special character as a regular character, as in "literally".
- A non-quoted backslash, \ is the Bash escape character.
  - It preserves the literal value of the next character that follows, with the exception of newline.
  - If a `\newline` pair appears, and the backslash itself is not quoted, the `\newline` is treated as a line continuation (that is, it is removed from the input stream and effectively ignored).

Single Quotes

- Enclosing characters in single quotes, `''` preserves the literal value of each character within the quotes.
- A single quote may not occur between single quotes, even when preceded by a backslash.
- IMPORTANT: Variables are not replaced with their values when referenced within single quotes.

```
FOO='$NAME'
echo $FOO
$NAME
```

Double Quotes

- Enclosing characters in double quotes, " " preserves the literal value of all characters within the quotes, with the exception of $, `, and \.
  - A double quote may be quoted within double quotes by preceding it with a backslash.
  - The characters $ and ` retain their special meaning within double quotes, i.e., parameter expansion and command substitution.

```
A=test
echo "$A"
test
echo "`ls`"
1.txt 2.txt 3.txt
```

Alternative Command Substitution

- As an alternative to the backticks, you can use parentheses for command substitution
- See: https://www.gnu.org/software/bash/manual/html_node/Command-Substitution.html
- Example:

```
echo "`ls`"
1.txt 2.txt 3.txt
echo "$(ls)"
1.txt 2.txt 3.txt
```

String Manipulation in Bash

Substring extraction:

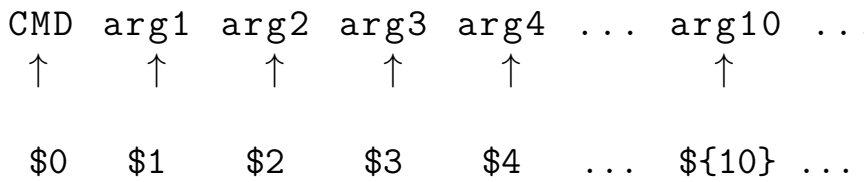
```
SUBSTR=${STR:5:3}
```

SUBSTR will contain 3 characters of STR starting at index 5.

See: https://tldp.org/LDP/abs/html/string-manipulation.html for more string manipulation functions.

Command Line Arguments

- They are the arguments specified at the command prompt with a command or script to be executed.
- Each argument is stored as a special shell variable for use in a command or script.



Other Built-in Variables

- Other variables related to command line arguments are:
- `$#` : The number of command line arguments supplied to the script
- `$*` : All the arguments treated as one double quoted string
- `$@` : All the arguments treated as individual double quoted strings. Can be used to loop through variables in `for` loops.
- `$$` : The process id of the current shell

Example

```
#!/bin/bash
echo "Script name: $0"
echo "Process: $$"
echo "First arg: $1"
echo "Second arg: $2"
echo "All args: $*"
```

if statements

if syntax

```
if TEST; then
     Bash commands ...
[elif TEST; then
     Bash commands ...]
[else
     Bash commands ...]
fi
```

- Use `if` to perform conditional branching in your Bash scripts
- The TESTs that can be performed can be listed using `man test`

Example

```
#!/bin/bash
if [ $1 -le 10 ]; then
    echo "Enter a number larger than 10"
fi
```

or

```
#!/bin/bash
if test $1 -le 10; then
    echo "Enter a number larger than 10"
fi
```

More on the `test` command

`test expression` OR `[ expression ]`

- The square brackets are an alternative to the `test` command and have to include space characters before and after
- The `test` command's exit status is used to determine which branch to take.
  - However, this could be very confusing to C programmers, because 0 is used to represent `false` in C; whereas a successful test, i.e., `true`, has exit status 0.

Tests

The `test` command provides various tests, which can be grouped into the following categories:

- Tests on files
- Tests on strings
- Tests on integers
- Combining tests with logical expressions

Tests on files

Below are some common tests on files. Check out the man page for more.

- `[ -e file ]` → true if file exists
- `[ -d file ]` → true if file exists and it's a directory
- `[ -r file ]` → true if file exists and it's readable
- `[ -s file ]` → true if file exists and its size is greater than 0
- `[ file1 -nt file2 ]` → true if file1 exists and it's newer than file2

Tests on strings

Below are some common tests on strings. Check out the man page for more.

- `[ s1 = s2 ]` → true if the strings are identical (Note the spaces around =)
- `[ s1 != s2 ]` → true if the strings are not identical
- `[ -n string ]` → true if string is not empty
- `[ -z string ]` → true if string is empty
- `[ s1 < s2 ]` → true if s1 comes before s2 based on the binary value of their characters

Tests on integers

Below are all the tests available for integers.

- `[ n1 -eq n2 ]` → true if the integers are equal
- `[ n1 -ne n2 ]` → true if the integers are not equal
- `[ n1 -gt n2 ]` → true if n1 is greater than n2
- `[ n1 -ge n2 ]` → true if n1 is greater than or equal to n2
- `[ n1 -lt n2 ]` → true if n1 is less than n2
- `[ n1 -le n2 ]` → true if n1 is less than or equal to n2

Logical operators

Tests can be combined with the following logical operators.

- `[ ! expression ]` → true if the expression is false
- `[ e1 -a e2 ]` → Logical AND operator. True if both of the expressions are true
- `[ e1 -o e2 ]` → Logical OR operator. True if either one of the expressions is true
- `( expression )` → true if expression is true. Can be used to group expressions within parentheses.

Example

```
#!/bin/bash
echo -n "Please enter a whole number: "
read VAR
echo Your number is $VAR
if [ $VAR -gt 100 ]; then
        echo "It's greater than 100"
elif [ $VAR -lt 100 ]; then
        echo "It's less than 100"
else
        echo "It's exactly 100"
fi
```

There is more ...

- The test syntax described in the previous slides are for the `test` command.
- Bash also provides additional *shell* functionality to
  - combine exit codes with `&& and ||`,
  - create subprocess environments with single parentheses
  - perform arithmetic evaluation and C-style variable manipulation with double parentheses
  - perform tests with C-style comparison operators with double-brackets
    - Read more at: https://www.baeldung.com/linux/bash-single-vs-double-brackets
  - More on advanced if statements: https://www.baeldung.com/linux/bash-single-vs-double-brackets

case ... esac

case syntax

```
  case "$variable" in
     pattern1)
        commands;;
     pattern2)
        commands;;
     *)
        commands;;
esac
```

Use case to perform different actions on different values of a variable.

case Example

```
case "$letter" in
   "e"|"o"|"a"|"i"|"u")
      echo "vowel";;
   *)
      echo "consonant";;
esac
```

for loops

for syntax

```
for VAR in 1 3 5 7 9
do
     Bash commands ...
done
```

- Use `for` loops to repeat Bash commands
- There are alternative ways to specify the values of a loop variable (will be discussed in the following slides)

Using the {START..END..INCREMENT} syntax:

Example:

```
for VAR in {1..10..2}
do
     echo $VAR
done
```

Using the output of a command:

Example:

```
for FILE in $(ls *.txt)
do
     echo $FILE
done
```

Using the C-style for loop:

Example:

```
for (( i=1; i<10; i+=2 ))
do
     echo $i
done
```

Looping over command line arguments:

Example:

```
for VAR in $@
do
     echo $VAR
done
```

`continue` and `break` commands

- You can continue to the next iteration by skipping over the statements in the rest of the for loop body using the `continue` command.
- You can break out of the for loop using the `break` command.

More on for loops

Read more at:

- https://www.cyberciti.biz/faq/bash-for-loop/

while loops

`while` syntax

```
while TEST
do
     Bash commands ...
done
```

- Use `while` loops to repeat Bash commands
- The TEST for the while loop is the same TEST we discussed for the if statements
- `continue` and `break` commands work as in `for` loops

More on while loops

- To increment a while loop variable, you can use the arithmetic evaluation environment, with double parentheses. Example:

```
#!/bin/bash
x=1
while [ $x -le 5 ]
do
   echo "Welcome $x times"
   x=$(( $x + 1 ))
done
```

- Read more at:
  - https://www.cyberciti.biz/faq/bash-while-loop/