CSCB09 Software Tools and Systems Programming
Shell Scripting

Marcelo Ponce

Winter 2023

Department of Computer and Mathematical Sciences - UTSC

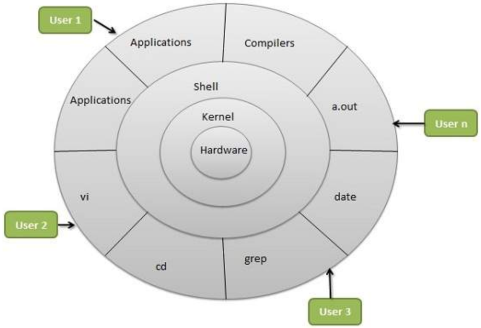
Today's class

Today we will discuss the following topics:

- The Shell
  Shell Programming/Scripting

Shell Programming

the shell

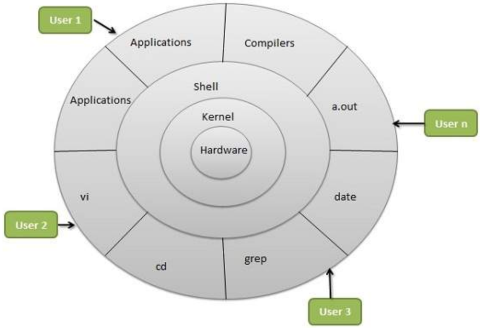


- a user interface to access OS's services
- interprete commands
- command line interface (CLI)
- access it using a console, terminal, CLI, ...

Multiple types of shells:

sh, bash, csh, ksh, tcsh, zsh

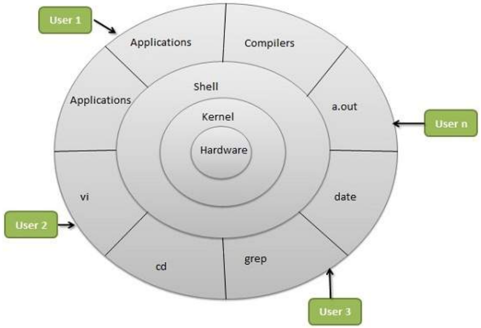
the shell



Multiple types of shells:
sh, bash, csh, ksh, tcsh, zsh

- a user interface to access OS's services
- interprete commands
- command line interface (CLI)
- access it using a console, terminal, CLI, ...
- list of accessible shells in a system,
  `cat /etc/shells`

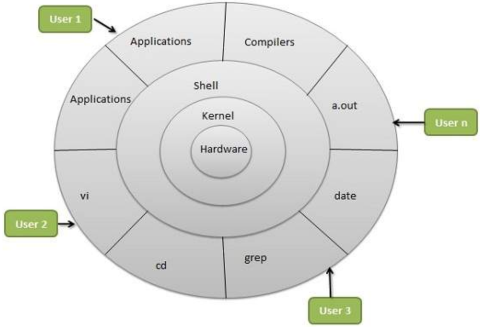
the shell



Multiple types of shells:
sh, bash, csh, ksh, tcsh, zsh

- a user interface to access OS's services
- interprete commands
- command line interface (CLI)
- access it using a console, terminal, CLI, ...
- list of accessible shells in a system,
  `cat /etc/shells`
- commands can be "bundled" together into a program $\rightsquigarrow$ **script**

the shell



Multiple types of shells:
sh, bash, csh, ksh, tcsh, zsh

- a user interface to access OS's services
- interprete commands
- command line interface (CLI)
- access it using a console, terminal, CLI, ...
- list of accessible shells in a system,
  `cat /etc/shells`
- commands can be "bundled" together into a program $\rightsquigarrow$ **script**
- More over the shell contains the elementary block of a programming language, i.e. conditionals, loops, functions, etc.

Shell Scritping

- The shell is not only a command interpreter, but also a fairly powerful programming language.
- A shell program, also called a script, is an easy-to-use tool for building applications.
- It can easily glue together tools, utilities, compiled binaries, system calls and other scripts.
- Shell scripting follows the classic UNIX philosophy of breaking complex projects into simpler subtasks and chaining together components and utilities.
- Shell scripts are well suited for administrative system tasks and any other routine repetitive tasks.
- In the simplest case, a script is nothing more than a list of system commands stored in a file.

Generalities

- Almost everything is treated as text strings
- Spaces matter regularly
- Remember the difference between return values and standard output
- Once you need a data-structure, use Python
- Commands run from a file in a *subshell*
- A great way to automate a repeated sequence of commands.

Commands

- You can run any program in a shell by calling it as you would on the command line.
- When you run a program like `grep` or `ls` in a shell program, a new process is created – recall `exec`.
- There are also some built-in commands where no new process is created:
  `echo exit set test read shift`

```
#!/bin/bash

echo "Hello␣World!"
```

- Any text following the "#" is considered a comment
- The `#!` line in a shell script will be the first thing the command interpreter sees.
- The `#!` is called the *sha-bang*, but more commonly seen in the literature as *she-bang* or *sh-bang*, as it derives from the concatenation of the tokens sharp (#) and bang (!), is actually a *two-byte magic number*, a special marker that designates a file type, or in this case an executable shell script.
- Immediately following the sha-bang is a path name to the program that interprets the commands in the script, e.g. "`/bin/bash`".
- Not the same on every machine!
- `#!` can be omitted if the script consists only of a set of generic system commands, using no internal shell directives.

Variables

- Shell variables are created once they are assigned a value.
- A variable can contain a number, a character or a string of characters.
- Variable name is case sensitive and can consist of a combination of letters and the underscore "_".
- Value assignment is done using the "=" sign.
- Assignment operator has no spaces.
- Loose convention: CAPS
- Dereference with $, i.e. `$VAR_NAME`

```
NAME="Marcelo"

CLASS="B09"

Year=2023
```

```
echo $NAME

Marcelo

echo $Year

2023
```

```
echo CLASS

CLASS

echo $CLASS

B09
```

- A backslash "\" is used to escape special character meaning:

```
PRICE_PER_APPLE=5
echo "The price of an apple today is:  \$ $PRICE_PER_APPLE"
The price of an apple today is:  $ 5
```

- A backslash "\" is used to escape special character meaning:

```
PRICE_PER_APPLE=5
echo "The price of an apple today is:  \$ $PRICE_PER_APPLE"
The price of an apple today is:  $ 5
```

- Encapsulating the variable name with $\boxed{\$}$ is used to avoid ambiguity:

```
MyFirstLetters=ABC
echo "The first 10 letters in the alphabet are:
${MyFirstLetters}DEFGHIJ"
The first 10 letters in the alphabet are:  ABCDEFGHIJ
```

- A backslash "\" is used to escape special character meaning:

```
PRICE_PER_APPLE=5
echo "The price of an apple today is:  \$ $PRICE_PER_APPLE"
The price of an apple today is:  $ 5
```

- Encapsulating the variable name with $\boxed{\$}$ is used to avoid ambiguity:

```
MyFirstLetters=ABC
echo "The first 10 letters in the alphabet are:
${MyFirstLetters}DEFGHIJ"
The first 10 letters in the alphabet are:  ABCDEFGHIJ
```

- Encapsulating the variable name with "" will preserve any white space values:

```
greeting='Hello world!'
echo $greeting
echo "Now with spaces:  $greeting"
Hello world!
Now with spaces:  Hello world!
```

Command Substitution

$\boxed{\texttt{`...`}}$ –bask-sticks– and $\boxed{\texttt{\$(...)}}$ cause command substitution

```
`do command and put stdout here`

$(do command and put stdout here)
```

E.g.

```
echo `ls`

echo $(ls)
```

Command Substitution

$\boxed{\texttt{`...`}}$ –bask-sticks– and $\boxed{\texttt{\$(...)}}$ cause command substitution

```
`do command and put stdout here`
```

```
$(do command and put stdout here)
```

E.g.

```
echo `ls`
```

```
echo $(ls)
```

Variables can be assigned with the value of a command output.

```
FILELIST=`ls`
```

```
FileWithTimeStamp=/tmp/my-dir/file_$(/bin/date +%Y-%m-%d).txt
```

Quoting

Double quotes inhibit wildcard replacement only.

Single quotes inhibit wildcard replacement, variable substitution and command substitution.

- `"` – double quotes
- `'` – single quote
- `` ` `` – back quote, aka back-sticks

```
echo Today is date
Today is date

echo Today is `date`
Today is Thu Sep 19 12:28:55 EST 2002

echo "Today is `date`"
Today is Thu Sep 19 12:28:55 EST 2002

echo 'Today is `date`'
Today is `date`
```

Arrays

- An array can hold several values under one name.
- An array is initialized by assign space-delimited values enclosed in $\boxed{()}$

```
my_array=(apple banana "Fruit Basket" orange)
new_array[2]=apricot
```

Arrays

- An array can hold several values under one name.
- An array is initialized by assign space-delimited values enclosed in $\boxed{()}$

```
my_array=(apple banana "Fruit Basket" orange)
new_array[2]=apricot
```

- The total number of elements in the array is referenced by `${#arrayname[@]}`

```
echo ${#my_array[@]}
```

4

- The array elements can be accessed with their numeric index.

  The first element in an array is element 0

```
echo ${my_array[3]} # orange - note that curly brackets are needed
my_array[4]="carrot" # adding a new element to the array
echo ${#my_array[@]}
```

5

Arithmetic Operations

Simple arithmetics on variables can be done using the arithmetic expression:

```
$((expression))
```

```
A=3

B=$((100 * $A + 5))

echo $B

305
```

Basic Operators:

- `a + b` addition (a plus b)
- `a - b` substraction (a minus b)
- `a * b` multiplication (a times b)
- `a / b` division (integer) (a divided by b)
- `a % b` modulo (the integer remainder of a divided by b)
- `a ** b` exponentiation (a to the power of b)

One can also use `expr`

Conditionals

The basic conditional decision making construct is:

```
if [ expression ]; then
  code if 'expression' is true
fi
```

Conditionals

The basic conditional decision making construct is:

```
if [ expression ]; then
  code if 'expression' is true
fi
```

```
NAME="Bard"
if [ "$NAME" = "Bard" ]; then
  echo "True - my name is indeed Bard"
fi
```

Conditionals

The basic conditional decision making construct is:

```
if [ expression ]; then
  code if 'expression' is true
fi
```

```
NAME="Bard"
if [ "$NAME" = "Bard" ]; then
  echo "True␣-␣my␣name␣is␣indeed␣Bard"
fi
```

It can be expanded with 'else'

```
NAME="Bing"
if [ "$NAME" = "Bard" ]; then
  echo "True␣-␣my␣name␣is␣indeed␣Bard"
else
  echo "False"
  echo "You␣have␣mistaken␣me␣for␣$NAME"
fi
```

Conditionals

The basic conditional decision making construct is:

```
if [ expression ]; then
  code if 'expression' is true
fi
```

```
NAME="Bard"
if [ "$NAME" = "Bard" ]; then
  echo "True␣-␣my␣name␣is␣indeed␣Bard"
fi
```

It can be expanded with '`else`'

```
NAME="Bing"
if [ "$NAME" = "Bard" ]; then
  echo "True␣-␣my␣name␣is␣indeed␣Bard"
else
  echo "False"
  echo "You␣have␣mistaken␣me␣for␣$NAME"
fi
```

It can be expanded with '`elif`' (else-if)

```
NAME="Mr.Data"
if [ "$NAME" = "Bard" ]; then
  echo "My␣parents␣were␣LambDas"
elif [ "$NAME" = "Bing" ]; then
  echo "I␣like␣to␣work␣in␣teams"
else
  echo "This␣leaves␣us␣with...␣"
fi
```

Comparisons

Numerical Comparisons

`$a -lt $b`	\$a < \$b
`$a -gt $b`	\$a > \$b
`$a -le $b`	\$a <= \$b
`$a -ge $b`	\$a >= \$b
`$a -eq $b`	\$a is equal to \$b
`$a -ne $b`	\$a is not equal to \$b

String Comparisons

`"$a" = "$b"`	\$a is the same as \$b
`"$a" == "$b"`	\$a is the same as \$b
`"$a" != "$b"`	\$a is different from \$b
`-z "$a"`	True is the length of "\$a" is zero
`-n "$a"`	True is the length of "\$a" is non-zero

Testing

The built-in command `test` is used to construct *conditional statements* in Bourne shell

`-d filename`	Exists as directory
`-f filename`	Exists as regular file
`-r filename`	Exists as readable
`-w filename`	Exists as writable
`-x filename`	Exists as executable
`-z string`	True if empty string
`str1 = str2`	True if `str1` equals `str2`
`int1 -eq int2`	True if `int1` equals `int2`
`-ne -gt -ge -lt -le`	$\neq \quad > \quad \geq \quad < \quad \leq$
`-a -o`	and or

More on `if`-statements

```
if test ! -d notes
then
  echo not found
else
  echo found
fi
```

```
if [ ! -d notes ] # spaces matters
then
  echo not found
else
  echo found
fi
```

`if` statements just check the return value of the command.

`test` is just a command that returns a value.

More on `if`-statements

```
if test ! -d notes
then
  echo not found
else
  echo found
fi
```

```
if [ ! -d notes ] # spaces matters
then
  echo not found
else
  echo found
fi
```

`if` statements just check the return value of the command.

`test` is just a command that returns a value.

```
if grep name file
then
  echo found
else
  echo not found
fi
```

Decision Making – case

case structure

```
case "$variable" in
  "$condition1" )
    command...
  ;;
  "$condition2" )
    command...
  ;;
esac
```

Decision Making – case

case structure

```
case "$variable" in
  "$condition1" )
    command...
  ;;
  "$condition2" )
    command...
  ;;
esac
```

simple case bash structure

```
mycase=1
case $mycase in
  1) echo "You␣selected␣bash";;
  2) echo "You␣selected␣perl";;
  3) echo "You␣selected␣phyton";;
  4) echo "You␣selected␣c++";;
  5) exit
esac
```

Loops: `for` loop

`for` **loop**
For each pass through the loop, arg takes on the value of each successive value in the list. Then the command(s) are executed.

```
# basic construct
for arg in [list]
do
  command(s)...
done
```

```
for i in 1 2 3 4; do
  echo $i
done
```

```
iters="1␣2␣3␣4"
for i in $iters; do
  echo $i
done
```

```
for i in foo bar
do
  echo hello there
done
```

```
for i in `seq 10`; do
  echo $i
done
```

```
files=`ls`
for f in $files; do
  cat $f
done
```

Loops: `while` loop

`while` loop
The `while` construct tests for a condition, and if true, executes commands.
It keeps looping as long as the condition is true.

```
# basic construct
while [ condition ]
do
  command(s)...
done
```

Loops: `while` loop

`while` loop

The `while` construct tests for a condition, and if true, executes commands.

It keeps looping as long as the condition is true.

```
# basic construct
while [ condition ]
do
  command(s)...
done
```

```
COUNT=4
while [ $COUNT -gt 0 ]; do
  echo "Value of count is: $COUNT"
  COUNT=$(($COUNT - 1))
done
```

Loops: `until` loop

`until` loop

The `until` construct tests for a condition, and if false, executes commands.

It keeps looping as long as the condition is false – opposite of the `while` construct.

```
until [ condition ]
do
  command(s)...
done
```

Loops: `until` loop

`until` loop

The `until` construct tests for a condition, and if false, executes commands.

It keeps looping as long as the condition is false – opposite of the `while` construct.

```
until [ condition ]
do
  command(s)...
done
```

```
COUNT=1
until [ $COUNT -gt 5 ]; do
  echo "Value␣of␣count␣is:␣$COUNT"
  COUNT=$(($COUNT + 1))
done
```

Loops – breaking the loops...

`break` and `continue` statements
`break` and `continue` can be used to control the loop execution of `for`, `while` and `until` constructs.

`continue` is used to skip the rest of a particular loop iteration, whereas `break` is used to skip the entire rest of loop

Loops – breaking the loops...

break and continue statements

break and continue can be used to control the loop execution of for, while and until constructs.

continue is used to skip the rest of a particular loop iteration, whereas break is used to skip the entire rest of loop

```
# Prints out 0,1,2,3,4
COUNT=0
while [ $COUNT -ge 0 ]; do
  echo "Value␣of␣COUNT␣is:␣$COUNT"
  COUNT=$((COUNT+1))
  if [ $COUNT -ge 5 ] ; then
    break
  fi
done
```

```
# Prints out only odd numbers -
    1,3,5,7,9
COUNT=0
while [ $COUNT -lt 10 ]; do
  COUNT=$((COUNT+1))
  # Check if COUNT is even
  if [ $(($COUNT % 2)) = 0 ] ; then
    continue
  fi
  echo $COUNT
done
```

Command-line Arguments

positional parameters
variables that are assigned according to position in a string

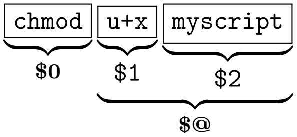
command line arguments are placed in positional parameters

$$\underbrace{\boxed{\texttt{chmod}}}_{\$0}\;\underbrace{\underbrace{\boxed{\texttt{u+x}}}_{\$1}\;\underbrace{\boxed{\texttt{myscript}}}_{\$2}}_{\$@}$$

Command-line Arguments

positional parameters
variables that are assigned according to position in a string

command line arguments are placed in positional parameters



Example

```
#!/bin/sh

echo arg1: $1
echo arg2: $2
echo name: $0
```

Positional Parameters

$0	Name of script
$#	Number of positional parameters
$*	Lists all positional parameters
$1 .. $9	First 9 positional parameters
$10	10th positional parameter

$*	Lists all positional parameters
$@	Same as $* except when in quotes
"$*"	Expands to single argument "$1 $2 $3"
"$@"	Expands to separate args "$1" "$2" "$3"

Positional Parameters

$0	Name of script
$#	Number of positional parameters
$*	Lists all positional parameters
$1 .. $9	First 9 positional parameters
$10	10th positional parameter

$*	Lists all positional parameters
$@	Same as $* except when in quotes
"$*"	Expands to single argument "$1 $2 $3"
"$@"	Expands to separate args "$1" "$2" "$3"

Iterating Over Arguments

```
#!/bin/sh
for arg in "$@"
do
   echo $arg
done
```

Positional Parameters

$0	Name of script
$#	Number of positional parameters
$*	Lists all positional parameters
$1 .. $9	First 9 positional parameters
$10	10th positional parameter

$*	Lists all positional parameters
$@	Same as $* except when in quotes
"$*"	Expands to single argument "$1 $2 $3"
"$@"	Expands to separate args "$1" "$2" "$3"

Iterating Over Arguments

```
#!/bin/sh
for arg in "$@"
do
   echo $arg
done
```

Arguments Default Values

```
arg1=${1:-def1}
arg2=${2:-"x"}
```

set and shift

set
assigns positional parameters to its arguments

```
$ set `date`
$ echo "The date today is $2 $3, $6"

The date today is May 29, 2023
```

shift
change the meaning of the positional parameters

```
while test "$1"
do
  echo $1
  shift
done
```

Shell Arithmetics – `expr`

Since variables work by text replacement, we need a special way to do arithmetic

```
x=1
x=$x + 1  # Nope!
x=$x+1    # Not that either!
x=`expr $x + 1`
y=`expr $x \* 5` #need to escape *
```

Functions i

A function is a subroutine that implements a set of commands and operations.

You can create your own functions or subroutines:

```
# basic construct
function_name {
  command...
}
```

```
myfunc() {
  arg1=$1
  arg2=$2
  echo $arg1 $globalvar
  return 0
}

globalvar="I am an evil global var :("
myfunc num1 num2
```

Functions ii

- Functions are called simply by writing their names.
- A function call is equivalent to a command.
- Parameters may be passed to a function, by specifying them after the function name.
- The first parameter is referred to in the function as $1, the second as $2, etc.

```
# Functions declarations
function function_B {
  echo "Function␣B."
}

function function_A {
  echo "$1"
}

function adder {
  echo "$(($1␣+␣$2))" # another way to
      perform arithmetics
}


# FUNCTION CALLS
# Pass parameter to function A
function_A "Function␣A."  # Function A
function_B       # Function B

# Pass two parameters to function adder
adder 12 56      # 68
```

Special Variables

These are some special variables in shell:

$0	The filename of the current script.
$n	The n-th argument passed to script was invoked or function was called.
$#	The number of argument passed to script or function.
$@	All arguments passed to script or function.
$*	All arguments passed to script or function.
$?	The exit status of the last command executed.
$$	The process ID of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	The process number of the last background command.

read

read one line from standard input and assigns successive words to the specified variables. Leftover words are assigned to the last variable.

```
# read from standar input
#!/bin/sh
echo "Enter␣your␣full␣name:"
read fName lName
echo "First:␣$fName"
echo "Last:␣$lName"
```

```
# read from a file
while read line
do
  echo $line
done < $file
```

Signal/Interrupt Trapping

It often comes the situations that you want to catch a special signal/interruption/user input in your script to prevent the unpredictables.

`trap` is the command to try:

```
trap <arg/function> <signal>
```

```
#!/bin/bash
# traptest.sh
# notice you cannot make Ctrl-C work in this
    shell,
# try with your local one, also remeber to chmod
     +x # your local .sh file so you can execute
     it!

trap "echo␣Booh!" SIGINT SIGTERM
echo "it's␣going␣to␣run␣until␣you␣hit␣Ctrl+Z"
echo "hit␣Ctrl+C␣to␣be␣blown␣away!"

while true:
do
  sleep 60
done
```

```
function booh {
  echo "booh!"
}

# And call it in a trap:
trap booh SIGINT SIGTERM
```

Bash trap Command

```
$ kill -l
 1) SIGHUP    2) SIGINT    3) SIGQUIT   4) SIGILL    5) SIGTRAP
 6) SIGABRT   7) SIGBUS    8) SIGFPE    9) SIGKILL  10) SIGUSR1
11) SIGSEGV  12) SIGUSR2  13) SIGPIPE  14) SIGALRM  15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN  22) SIGTTOU  23) SIGURG   24) SIGXCPU  25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH  29) SIGIO 30) SIGPWR
31) SIGSYS   34) SIGRTMIN   35) SIGRTMIN+1   36) SIGRTMIN+2   37) SIGRTMIN+3
38) SIGRTMIN+4   39) SIGRTMIN+5   40) SIGRTMIN+6   41) SIGRTMIN+7   42) SIGRTMIN+8
43) SIGRTMIN+9   44) SIGRTMIN+10  45) SIGRTMIN+11  46) SIGRTMIN+12  47) SIGRTMIN+13
48) SIGRTMIN+14  49) SIGRTMIN+15  50) SIGRTMAX-14  51) SIGRTMAX-13  52) SIGRTMAX-12
53) SIGRTMAX-11  54) SIGRTMAX-10  55) SIGRTMAX-9   56) SIGRTMAX-8   57) SIGRTMAX-7
58) SIGRTMAX-6   59) SIGRTMAX-5   60) SIGRTMAX-4   61) SIGRTMAX-3   62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX
```

Recall the numbers in front of each signal name, one can use that number to avoid typing long strings in trap:

```
#2 corresponds to SIGINT and 15 corresponds to SIGTERM
trap booh 2 15
```

Other relevant topics

- Redirection and piping: `>` `>>` `<` `|`
- Multiple useful commands
- Linux file system & directory structure
- Super-powerful tools: `awk`, `sed`, ...