Note: you can find all slides of this tutorial under:

`http://www.cs.queensu.ca/~acmteam/advunix.pdf`

The introductory Unix tutorial can be found under:

`http://www.cs.queensu.ca/~acmteam/unix.pdf`

Advanced Unix Tutorial

In this tutorial, you will learn about:

- Common Unix tools (grep, sed, awk, tr, etc.)
- Environment variables
- Csh/tcsh basics
- Csh/tcsh Shell scripts

More Useful Commands

grep/egrep - searches lines for patterns using regular expressions.

```
grep [options] [pattern] [file ...]
```

E.g. To print all lines that contain `double` in all `*.java` files:

```
grep double *.java
```

Useful options:

`-i` Case-insensitive search

`-v` Reverse search (print all lines that do not contain the pattern)

`-n` Add line number to the lines found

E.g. To print all lines that do not contain `system` (case-insensitive) in `*.java`:

```
grep -iv system *.java
```

Regular Expressions — a string that represents multiple instances

It can be used with **egrep** for pattern search. (**man -s 5 regexp**)

Examples: **egrep "a[x-z]c" file1 file2**

Pattern	Matches
a.c	a[*any single character*]c, e.g. abc, a1c, a c.
a[xyz]c or a[x-z]c	axc, ayc, and azc only
a[^xyz]c	a[*any single character but x, y, or z*]c
ab*c	a[*0 or more* b]c, e.g. ac, abc, abbbc
ab+c	a[*1 or more* b]c, e.g. abc, abbbc
^abc	abc only at the beginning of a line
abc$	abc only at the end of a line
a(bc|de)f	abcf and adef
myarray\[.+\]	myarray[*anything that has 1 or more character*]

cut — select a list of columns or fields from one or more files

Fields and columns start at 1.
Example: (myfile is a file of `abcdefghijklmnopqrstuvwxyz`)

- To see only the second and forth character of file `myfile`:

  `cut -c2,4 myfile` (output = `bd`)

- For characters from 1st to 3rd, 10th to 12th, 24th to the end:

  `cut -c-3,10-12,24- myfile` (output = `abcjklxyz`)

- Cut can also display fields split by a delimiter (separator):

  `echo "12#34#567#8" | cut -d"#" -f2-3` (`34#56`)

- Find out who is logged on, but list only usernames:

  `who | cut -d " " -f1`

tr - translate characters

`tr` copies standard input to standard output, substituting or deleting specified characters, for example:

```
tr A-LM-Z a-z < file1 > file2
```

creates `file2` as a copy of `file1`, with all uppercase letters translated to the corresponding lowercase ones.

`tr str1 str2`	translates `str1` chars to the corresponding `str2`
`tr -s str1 str2`	squeezes repeated chars in `str1` to 1 char
`tr -d str1`	removes all chars in `str1`

There are more sophisticated uses of `tr` which are very useful, e.g., `tr -s '[:blank:]' '[\012*]'` changes each set of whitespaces to a single newline (`\012` is newline in octal).

The Shell

- The user interface of Unix is the shell
- Some UNIX workstations offer GUIs to enhance the user interface
- Within a window the shell remains the control center
- Several shells are available: sh (Bourne Shell), ksh (Korn Shell), csh, and tcsh
- We will be looking at tcsh (tcsh is an enhanced version of csh), and we will use the word csh and tcsh interchangeably

Environment Variables

- Unix keeps user-defined shell environment parameters (user info and preferences) in environment variables
- Environment variables constitute the environment of the shell
- `HOME` — variable representing your home directory, e.g., `printenv HOME` or `echo $HOME` shows your home directory
- `PATH` — the list of directories that form the command search path, e.g.

  `setenv PATH $HOME/bin:$PATH` (add to .cshrc file)

  tells the shell to look in the users home directory under the bin directory for commands
- Use `printenv` to see your environment variables

Environment Variables and Shell Variables

- Shell variables are variables for a particular shell. Unlike environment variables, shell variables won't be inherited to shells opened by the current shell
- Usually, environment variable names consist of uppercase letters, and shell variables consist of lowercase letters

	Environment Variables	Shell Variables
Assignment or define	`setenv` *name content* E.g. `setenv FOO bar`	`set` *name*`=`*content* E.g. `set foo=bar`
Remove	`unsetenv` *name*	`unset` *name/pattern*

- Trying to access an undefined variables (except for `unset`) will give you an error.

Environment and Shell Variables (cont'd)

- Shell variables can have arrays of 1D. Parentheses must be used to enclose the contents, which are separated by spaces:

  ```
  set myarray=(this is an array)
  ```

- Use square brackets to access element(s) of the array (1-based)
- To see all defined shell variables, use `set`

Some environment/shell variables defined automatically:

`$PATH` or `$path`	Directories to search for commands
`$HOME` or `$home`	User's home directory
`$noclobber`	If defined, prevents redirections (`>`) to overwrite files
`$prompt`	Control the appearance of the prompt
`$status`	The exit value of the previous command

Variable Operation	Description
`$`*name*`[`*i*`]` E.g. `echo $myarray[2]` E.g. `set $myarray[2]=was`	Access the i^{th} element Outputs `is` Changes `is` to `was`
`$`*name*`[`*i*`-`*j*`]` E.g. `echo $myarray[2-3]` E.g. `echo $myarray[2-]`	Access the i^{th} thru j^{th}element Output `was an` Output `was an array`
`$#`*name* (shell var only) `$#myarray`	Show the number of elements Output `4`
`$?`*name* E.g. `echo $?myarray`	Check if variable *name* is defined Output `1`
shift *name* (shell var only) E.g. `shift myarray`	Remove the first element of an array `$myarray` becomes `(was an array)`

Shell Variables — Arithmetic Operations

- Arithmetic operation must be performed using `@`:

  `@ var=expr` (note the space after `@`)

  `@ var[n]=expr`

- Only operations involving arithmetic needs `@`, for other operations use `set`

- Integers only (no floating point numbers)

Examples:

```
@ i = 10      (same as set i=10)
@ j = $#path / 2      (note the spaces around /)
@ myintarray[$j] = $j + 4
@ x += 3
@ i++
```

Arithmetic and bitwise logical operators

`+`	plus
`-`	minus
`*`	multiplication
`/`	division
`%`	modulus

`!`	not
`&`	bitwise and
`|`	bitwise or
`^`	bitwise exclusive-or
`<<`	left-shift
`>>`	right-shift

Note that an operator symbol must be surrounded by space:

```
@ a = $b % $c
```

Shell Scripting Intro

- The shell is not only a command interpreter, it also defines a simple programming language
- A program written in this language is called a shell script
- Shell scripts can save you a lot of time if you find yourself repeating commands over and over again
- Shell scripts are like batch files in DOS
- You can also type out all lines in a shell script at the prompt to do the same thing as the script

Shell Script Basics

- A shell script file starts with a line like this:
  `#!/usr/local/bin/tcsh`
  It indicates which command is used to interpret this script
- Consists of lines of commands
- Comments are preceded by #
- If the execution of a script results in an error, script execution is aborted if the command is built-in or skipped if the offending command isn't built-in
- A shell script file must have its readable and executable flags set in order to be run directly:
  `chmod a+rx myshellscript` (readable/executable for all)
  `myshellscript` (execute this script if it is in the path)

Passing Arguments

- Arguments can be passed to a tcsh script:
  `./myshellscript a1 b2 c3`
- Arguments are stored in the array variable `$argv`
- Alternatively, `$1` represents the first argument, `$2` the second etc.
- `$*` is equivalent to `$argv` (which is `a1 b2 c3`)
- `$0` is the command that runs the current script file (which is `./myshellscript`)
- `$argv[0]` is undefined

`foreach` loop

`foreach` allows one to execute a series of lines of commands for each of the element in a list:

`foreach` *index_variable_name* `(` *element element ...* `)`

command (can be `break` or `continue`)

...

`end`

```
#!/bin/csh
# list all files end with .java and .c
foreach file (*.java *.c)
   echo $file
end
```

if statement

```
if (condition) then
      ...
else if (condition) then
      ...
else
      ...
endif
```

Examples of conditions (also called expressions)

`($1 == $2)`	if the first arg is same as the second arg
`!($1 > $2)`	not `($1 > $2)`
`(-f file)`	if `file` is a file (not directory)
`(-d file)`	if `file` is a directory

Relational Operators

`==`	equal
`!=`	not equal
`>`	numerical greater than
`<`	numerical less than
`>=`	numerical greater than or equal to
`<=`	numerical less than or equal to
`=~`	string match (right side can be a pattern)
`!~`	not a string match

Example

```
if ($1 =~ m*)  echo "$1 starts with m"
```

Expressions

Logical Operators:

`||`	logical or
`&&`	logical and
`!`	logical not

Some file conditions, e.g. `if (-r filename) ...`

`(-r filename)`	True if `filename` is readable
`(-w filename)`	True if `filename` is writable
`(-x filename)`	True if `filename` is executable
`(-e filename)`	True if `filename` exists
`(-o filename)`	True if the user owns `filename`

```
#!/bin/csh
# Find the location of given command in the path
# Simulate the "which" command.
if ($#argv != 1) then
    echo "Usage:  $0 command"
    exit 1
endif
foreach dir ($path)
    set file=$dir/$1
    if (-f $file && -x $file) then
        echo "Found:  $file"
        exit 0
    endif
end
echo $1 not found
exit 1
```

`switch` statement

- similar to C or Java's `switch`

Example

```
#!/bin/csh
# append $1 to $2, or append standard input to $1
switch ($#argv)
  case 1:
          cat >> $argv[1]
          breaksw
  case 2:
          cat >> $argv[2] < $argv[1]
          breaksw
  default:
          echo 'usage: append [from] to '
endsw
```

`while` loop

- similar to while loop in C or Java
- `break` and `continue` can be used

```
#!/bin/csh
# Generate output files from input files
# Good for testing your program
set max=8
set i=1
while ($i <= $max)
   set infile=myInputFile.$i
   set outfile=myOutputFile.$i
   echo "To run with $infile, output to $outfile"
   java prog < $infile >&! $outfile # forces overwrite
   @ i++
end
```

Quotes

- There are three kinds of quotes: single ', double ", and back `
- Single and double quotes can be used to enclose a string
- Single quotes don't expand the string inside (i.e. leave the string as it is), double quotes do (i.e. return the contents of variables):
  `echo '$user'` outputs `$user`
  `echo "$user"` outputs `ttang`
- Backquotes evaluate the string enclosed:
  `echo "the command more is at `which more`"` outputs
  `the command more is at /usr/bin/more`

`awk` and `sed`

- They are standard Unix commands for text processing that can have scripts
- Nowadays people usually use *Perl* for text processing
- They are handy for simple operations:

```
awk '{print $1$3, $NF} myfile'
```

prints the 1st and 3rd (no space in between), and the last field of each line in `myfile`; and

```
sed "s/foo/bar/g;s/if/in case/" myfile
```

changes all occurrences of "foo" to "bar", and only the first occurrence of "if" to "in case"

Example - Simulate `move` in DOS (`mv *.txt *.doc` doesn't work)

```
#!/bin/csh
if ($#argv < 3) then
  echo "Usage:  $0  search_pat  replace_pat  file ..."
  echo "Example:  $0 '\.txt"'$'"' '\.doc' "'*.txt'
  exit
endif
set search=$1
set replace=$2
foreach file ($argv[3-])
  set newname=`echo $file | sed "s/$search/$replace/"`
  if ($file != $newname) then
    echo "Changing $file to $newname"
    mv $file $newname
  endif
end
```

Alias Substitution

- Alias allows you to redefine existing command name with a name of your own. Examples:

`alias h history`	use `h` as an abbreviation of `history`
`alias dir ls`	use `dir` as an abbreviation of `ls`
`alias ls 'ls -F'`	the switch `-F` will be used whenever `ls` is used
`alias rm 'rm -i'`	confirmation needed before removing a file

- Use `unalias` to remove an alias, e.g., `unalias ls`
- Use a backslash before an aliased command to temporarily unalias that command: `\rm *` will delete all files in the current directory without asking (dangerous, make sure you know what you are doing)
- Aliases are usually put in the file `~/.cshrc`

Configuring your tcsh

- The file `~/.cshrc` contains your configuration of csh/tcsh
- Some content of `.cshrc` may be depended to the system configuration. Your current `.cshrc` is probably written by your system administrator.
- You can put your own configuration in some file, say `~/.mycshrc`, and put the line:

  ```
  source ~/.mycshrc
  ```

  at the end of `.cshrc` to tell csh to load your configuration file
- The command `source` can also be used in the shell

Final Words

- We have introduced the basics of Unix and shell programming
- For serious shell programming, C shell is not the best choice:
  - for instance, C shell does not have subroutines
  - we suggest Bourne shell (sh/bash), or Korn shell (ksh)
- For serious text processing, Perl is the language to use
  - it is heavily used in WWW programming