

Sistemas Operativos Shell Scripts

Ricardo Ruiz Rodríguez

Instituto de Computación
Universidad Tecnológica de la Mixteca
Primavera 2014

Índice I

1 Input/Output

- Output
- Input
- File descriptors

2 Functions

- Introduction
- Creating and using functions

3 Text filters

- The head and tail commands
- Using grep
- Counting words

Introduction

- Until now you have been looking at commands that print out messages.
- When a command prints a message, the message is called output.
- Now you will look at the different types of output available to shell scripts, and the mechanisms used to obtain input from users.
- Specifically, the areas that you will cover are:
 - ① Output to the screen.
 - ② Output to a file.
 - ③ Input from a file.
 - ④ Input from users.

Output

- When a command produces output that is written to the terminal, we say that the program has printed its output to the **Standard Output**, or STDOUT.
- Error messages are not written to STDOUT, but instead they are written to a special type of output called **Standard Error** or STDERR, which is reserved for error messages.
- Most commands use STDERR for error messages and STDOUT for informational messages.
- You will look at how shell scripts can use STDOUT to output messages to each of the following:
 - ❶ The terminal (STDOUT).
 - ❷ A file.
 - ❸ The terminal and a file.

Output to the terminal I

- Two common commands print messages to the terminal (STDOUT):
 - 1 `echo`.
 - 2 `printf`.
- The **echo** command is mostly used for printing strings that require simple formatting.
- The **printf** command is the shell version of the C language function `printf`. It provides a high degree of flexibility in formatting output.
- The most common command used to output messages to the terminal is the **echo** command. Its syntax is:

echo *string*

Here *string* is the string you want printed.

Output to the terminal II

- The default behavior of `echo` is to add a newline at the end of its output.
- When you are generating a prompt, this is not the most user-friendly behavior. When the `c` escape sequence is used, `echo` does not output a newline when it finishes printing its input string.
- As an example of its use, this excerpt from a shell script:

```
echo -e "Making directories, please wait...\t\c"  
for i ${DIRS_TO_MAKE} ; do mkdir -p $i ; done  
echo "Done."
```

produces diagnostic output that looks like the following:

Output to the terminal III

Making directories, please wait... Done.

- Another possible use is shown in the following example:

```
echo -e "Copying files, please wait\t\c"
for i in ${FILES} ;
do
    cp $i $DEST && echo -e ".\c"
done
echo "\tDone."
```

The output is similar to the following:

Copying files, please wait Done

Output to the terminal IV

Here a single . is printed for each file that is copied.

printf |

- The **printf** command is similar to the echo command, in that it enables you to print messages to STDOUT.
- In its most basic form, its usage is identical to echo.
- The only major difference is that the string specified to printf explicitly requires the `\n` escape sequence at the end of a string, in order for a newline to print. The echo command prints the newline automatically.
- The power of printf comes from its capability to perform complicated formatting by using format specifications.

printf II

- The basic syntax for this is:

printf format arguments

Here, *format* is a string that contains one or more of the formatting sequences, and *arguments* are strings that correspond to the formatting sequences specified in *format*.

- For those who are familiar with the C language `printf` function, the formatting sequences supported by the **printf** command are identical.
- The formatting sequences have the form:

`%[-]m.nx`

Here `%` starts the formatting sequence and `x` identifies the formatting sequences type.

- Depending on the value of `x`, the integers *m* and *n* are interpreted differently.

printf IV

```
elif [ -h $i ]; then
    echo "symbolic link"
elif [ -f $i ]; then
    echo "file"
else
    echo "unknown"
fi
done
```

This script produces a table that lists all the visible files in the current directory along with their file type. The output looks similar to the following:

printf V

File Name	Type
RCS	directory
dev	directory
humor	directory
images	directory
index.html	file
install	directory
java	directory

- As you can see, the items in the table's rows are not lined up with the table headings.
- You could fix this using spaces and tabs in conjunction with the echo command, but using the printf command makes the task extremely easy.

printf VI

- After test the script, please remove the - character to the first format sequence, save the script and run it again.

printf VII

```
#!/bin/sh
printf "%-32s %s\n" "File Name" "File Type"
for i in * ;
do
    printf "%-32s " "$i"
    if [ -d "$i" ]; then
        echo "directory"
    elif [ -h "$i" ]; then
        echo "symbolic link"
    elif [ -f "$i" ]; then
        echo "file"
    else
        echo "unknown"
    fi;
done
```

printf VIII

done

Output redirection I

- In the process of developing a shell script, you often need to capture the output of a command and store it in a file.
- When the output is in a file, you can edit and modify it easily.
- The process of capturing the output of a command and storing it in a file is called **output redirection** because it redirects the output of a command into a file instead of the screen.
- To redirect the output of a command or a script to a file, instead of STDOUT, use the output redirection operator, `>`, as follows:

```
command > file  
list > file
```

Output redirection II

The first form redirects the output of the specified *command* to a specified *file*, whereas the second redirects the output of a specified *list* to a specified *file*. If file exists, its contents are overwritten; if file does not exist, it is created.

- Examples:

```
date > now
$ cat now
Mon Jun  3 13:32:17 CDT 2013
{ date; uptime; who ; } > mylog
```

- The shell provides a second form of output redirection with the `>>` operator, which appends output to a file. The basic syntax is:

Output redirection III

```
command >> file  
list >> file
```

In these forms, output is appended to the end of the specified *file*, or the specified *file* is created if it does not exist.

- Example:

```
{ date; uptime; who ; } >> mylog
```

Redirecting output to a file and the screen |

- In certain instances, you need to direct the output of a script to a file and onto the terminal.
- An example of this is shell scripts that are required to produce a log file of their activities.
- For interactive scripts, the log file cannot just contain the script's output redirected to a file.
- To redirect output to a file and the screen, use the **tee** command. The basic syntax is as follows:

command | tee file

Here *command* is the name of a command, such as `ls`, and *file* is the name of the file where you want the output written.

- For shell scripts that require all their output to be logged, the following `if` statement is often used:

Redirecting output to a file and the screen II

```
if [ "$LOGGING" != "true" ] ; then
    LOGGING="true" ; export LOGGING ;
    exec $0 | tee $LOGFILE
fi
```

- Here you check to see whether a variable, `$LOGGING`, indicates that logging is turned on. If it is, the script continues; otherwise, the script reruns, and **tee** sends the output to a log file.
- To record all the output from a script, this if statement is usually one of the first commands in a script.

Input

- Many UNIX programs are interactive and read input from the user.
- To use such programs in shell scripts, you need to provide them with input in a noninteractive manner.
- Also, scripts often need to ask the user for input in order to execute commands correctly.
- To provide input to interactive programs or to read input from the user, you need to use input redirection. Now, you will look at the following two methods in detail:
 - ① Input redirection from files.
 - ② Reading input from a user.
 - ③ Redirecting the output of one command to the input of another.

Input redirection

- When you need to use an interactive command in a script, you need to provide the command with input.
- One method for doing this is to store the input of the command in a file and then tell the command to read input from that file. You accomplish this using input redirection.
- The input can be redirected in a manner similar to output redirection. In general, input redirection is:

command < file

Here the contents of *file* become the input for *command*.

- For example, the following would be an excellent use of redirection:

```
mail rruiz@mixteco.utm.mx < Exam_Answers
```

Here the input to the mail command, which becomes the body of the mail message, is the file *Exam_Answers*.

Here documents |

- An additional use of input redirection is in the creation of **here documents**.
- A common use of here documents is in the generation of email messages within scripts and in the generation of files containing the values of all the variables in the script.
- Also, here documents store temporary information.
- Say you need to send a list of phone numbers or URLs to the printer. By using a here document, you can enter the information that you want to send to the printer into the here document and then send that here document to the printer. This is much simpler than using a temporary file, which needs to be created and then deleted.
- The general form for a here document is:

Here documents II

```
command << delimiter  
document  
delimiter
```

- Here the shell interprets the << operator as an instruction to read input until it finds a line containing the specified *delimiter*.
- All the input lines up to the line containing the delimiter are then fed into the standard input of the command.
- The delimiter tells the shell that the here document has completed. Without it, the shell continues to read input forever.
- The delimiter must be a single word that *does not* contain spaces or tabs.

Here documents III

- For example, to print a quick list of URLs, you could use the following here document:

```
lpr << URLs
http://www.utm.mx/~rruiz/
http://www.cisco.com/
http://www.linux.org/
http://www.gnu.org/
URLS
```

- You can also combine here documents with output redirection as follows:

Here documents IV

```
command > file << delimiter  
document  
delimiter
```

If used in this form, the output of *command* is redirected to the specified *file*, and the input of *command* becomes the here document.

- For example, you can use the following command to create a file with the short list of URLs given previously:

Here documents V

```
cat > urls << URLs
http://www.utm.mx/~rruiz/
http://www.cisco.com/
http://www.linux.org/
http://www.gnu.org/
URLS
```

Reading user input I

- A common task in shell scripts is to prompt users for input and then read their responses.
- To do this, use the **read** command to set the value of a variable and then evaluate the value of the variable with a **case** statement.
- The **read** command works as follows:

read name

It reads the entire line of user input until the user presses return and makes that line the value of the variable specified by *name*.
- An example of this is:

Reading user input II

```
YN=yes
printf "Dr. Gordon, do you want to play a game [$YN]? "
read YN
: ${YN:=yes}
case $YN in
    [yY]|[yY][eE][sS]) exec sawgame ;;
    *) echo "Maybe later." ;;
esac
```

- A common use of input redirection in conjunction with the **read** command is the reading of a file one line at a time using the while loop.
- The basic syntax is:

Reading user input III

```
while read LINE
do
    : # manipulate file here
done < file
```

- In the body of the while loop, you can manipulate each line of the specified file. A simple example of this is:

```
while read LINE
do
    case $LINE in
        *root*) echo $LINE ;;
    esac
done < /etc/passwd
```

Reading user input IV

- Here only the lines that contain the string root in the file `/etc/passwd` are displayed.

Pipelines I

- Most commands in UNIX that are designed to work with files can also read input from STDIN.
- This enables you to use one program to filter the output of another.
- This is one of the most common tasks in shell scripting: having one program manipulate the output of another program.
- You can redirect the output of one command to the input of another command using a **pipeline**, which connects several commands together with pipes as follows:

```
command1 | command2 | ...
```

The pipe character, `|`, connects the standard output of `command1` to the standard input of `command2`, and so on.

Pipelines II

- The commands can be as simple or complex as are required.
- Examples of pipeline commands:

```
$ tail -f /var/adm/messages | more  
$ ps -ael | grep "$UID" | more
```

- In the first example, the standard output of the **tail** command is piped into the standard input of the **more** command, which enables the output to be viewed one screen at a time.
- In the second example, the standard output of **ps** is connected to the standard input of **grep**, and the standard output of **grep** is connected to the standard input of **more**, so that the output of **grep** can be viewed one screen at a time.

Pipelines III

- One important thing about pipelines is that each command is executed as a separate process, and the exit status of a pipeline is the exit status of the last command.
- It is vital to remember this fact when writing scripts that must do error handling.

Overview I

- When you issue any command, three files are opened and associated with that command.
- In the shell, each of these files is represented by a small integer called a **file descriptor**.
- A file descriptor is a mechanism by which you can associate a number with a filename and then use that number to read and write from the file.
- Sometimes file descriptors are called *file handles*.
- The three files opened for each command along with their corresponding file descriptors are:
 - ① Standard Input (STDIN), 0.
 - ② Standard Output (STDOUT), 1.
 - ③ Standard Error (STDERR), 2.

Overview II

- The integer following each of these files is its file descriptor.
- Usually, these files are associated with the user's terminal, but they can be redirected into other files.

Associating files with a file descriptor I

- By default, the shell provides you with three standard file descriptors for every command.
- With it, you can also associate any file with file descriptors using the **exec** command.
- Associating a file with a file descriptor is useful when you need to redirect output or input to a file many times but you don't want to repeat the filename several times.
- To open a file for writing, use one of the following forms:

exec *n*>file

exec *n*>>file

Here *n* is an integer, and *file* is the name of the file you want to open for writing.

Associating files with a file descriptor II

- The first form *overwrites* the specified file if it exists. The second form *appends* to the specified file.
- For example, the following:

```
$ exec 4>fd4.out
```

associates the file fd4.out with the file descriptor 4.

- To open a file for reading, you use the following form:

```
exec n<file
```

Here *n* is an integer, and *file* is the name of the file you want to open for reading.

General Input/Output redirection I

- You can perform general output redirection by combining a file descriptor and an output redirection operator.

- The general forms are:

command *n*> file

command *n*>> file

Here *command* is the name of a command, such as `ls`, *n* is a file descriptor (integer), and *file* is the name of the file.

- The first form redirects the output of *command* to the specified *file*, whereas the second form appends the output of *command* to the specified *file*.
- For example, you can write the standard output redirection forms in the general form as:

General Input/Output redirection II

command 1> file

command 1>> file

Here the *1* explicitly states that STDOUT is being redirected into the given file.

- General input redirection is similar to general output redirection. It is performed as follows:

command *n*<file

Here *command* is the name of a command, such as *ls*, *n* is a file descriptor (integer), and *file* is the name of the file.

- For example, the standard input redirection forms can be written in the general form as:

command 0<file

Redirecting STDOUT and STDERR to separate files I

- One of the most common uses of file descriptors is to redirect STDOUT and STDERR to separate files.
- The basic syntax is:

command 1> file1 2> file2

Here the STDOUT of the specified command is redirected to *file1*, and the STDERR (error messages) is redirected to *file2*.

- Often the STDOUT file descriptor, 1, is not written, so a shorter form of the basic syntax is:

command > file1 2> file2

- You can also use the append operator in place of either standard redirect operator:

Redirecting STDOUT and STDERR to separate files II

command >> file1 2> file2

command > file1 2>> file2

command >> file1 2>> file2

- The first form appends STDOUT to *file1* and redirects STDERR to *file2*.
- The second form redirects STDOUT to *file1* and appends STDERR to *file2*.
- The third form appends STDOUT to *file1* and appends STDERR to *file2*.
- Example:

```
for FILE in $FILES  
do
```

```
    ln -s $FILE ./docs >> /tmp/ln.log 2> /dev/null  
done
```

Redirecting STDOUT and STDERR to separate files III

Here the STDOUT of `ln` is appended to the file `/tmp/ln.log`, and the STDERR is redirected to the file `/dev/null`, in order to discard it.

- The file `/dev/null` is a special file available on all UNIX systems used to discard output. It is sometimes referred to as the **bit bucket**.

Redirecting STDOUT and STDERR to the same file

- Sometimes you need to redirect STDOUT and STDERR to the same file. In general, you do this by:

command > file 2>&1

list > file 2>&1

Here STDOUT (file description 1) and STDERR (file descriptor 2) are redirected into the specified *file*.

- Here is a situation where it is necessary to redirect both the standard output and the standard error:

```
rm -rf /tmp/my_tmp_dir > /dev/null 2>&1 ;  
mkdir /tmp/my_tmp_dir
```

Redirecting STDOUT and STDERR to the same file II

Here, you are not interested in the error message or the informational message printed by the `rm` command. You only want to remove the directory, thus its output or any error message it prints are redirected to `/dev/null`.

- If you had one command that should append its standard error and standard output to a file, you use the following form:

command >> file 2>&1

list >> file 2>&1

Redirecting redirecting... I

- You can also use output redirection to output error messages on STDERR.
- The basic syntax is:
echo string 1>&2 **printf** format args 1>&2
- You might also see these commands with the STDOUT file descriptor, 1, omitted:
echo string >&2 **printf** format args >&2
- As an example, say that you need to display an error message if a directory is given instead of a file. You can use the following if statement:

Redirecting redirecting... II

```
if [ ! -f $FILE ] ; then
    echo "ERROR: $FILE is not a file" >&2 ;
fi
```

- You can redirect STDOUT and STDERR to a single file by using the general form for redirecting the output of one file descriptor to another:
$$n > \&m$$

Here n and m are file descriptors (integers).

- If you let $n=2$ and $m=1$, you see that STDERR is redirected to STDOUT. By redirecting STDOUT to a file, you also redirect STDERR.

Redirecting redirecting... III

- If m is a hyphen (-) instead of a number, the file corresponding to the file descriptor n is closed. When a file descriptor is closed, trying to read or write from it results in an error.
- One of the most common uses of this form of redirection is for reading files one line at a time.
- You already looked at using a while loop to perform this task:

```
while read LINE
do
    : # manipulate file here
done < file
```

Redirecting redirecting... IV

- The main problem with this loop is that it is executed in a subshell, thus changes to the script environment, such as exporting variables and changing the current working directory, does not apply to the script after the while loop changes. As an example, consider the following script:

```
#!/bin/sh
if [ -f "$1" ] ; then
    i=0
    while read LINE
    do
        i=`echo "$i + 1" | bc`
    done < "$1"
    echo $i
fi
```

Redirecting redirecting... V

- This script *tries* to count the number of lines in the file specified to it as an argument.
- When the while loop exits, the value of `$i` is not preserved.
- In this case, you need to change a variable's value inside the while loop and then use that value outside the loop.
- You can accomplish this by redirecting the STDIN prior to entering the loop and then restoring STDIN to the terminal after the while loop.
- The basic syntax is:

Redirecting redirecting... VI

```
exec n<&0 < file
while read LINE
do
    : # manipulate file here
done
exec 0<&n n<&-
```

Here *n* is an integer greater than 2, and *file* is the name of the file you want to read.

- Usually *n* is chosen as a small number such as 3, 4, or 5.
- As an example, you can construct a shell version of the cat command:

Redirecting redirecting... VII

```
#!/bin/sh
if [ $# -ge 1 ] ; then
  for FILE in $@
  do
    exec 5<&0 < "$i"
    while read LINE ; do echo $LINE ; done
    exec 0<&5 5<&-
  done
fi
```

Introduction

- Shell functions provide a way of mapping a name to a list of commands.
- Shell functions are similar to subroutines, procedures, and functions in other programming languages.
- Think of them as miniature shell scripts that enable a name to be associated with a set of commands.
- The main difference is that a new instance of the shell begins in order to run a shell script, whereas functions run in the current shell.

Creating and using functions I

- The formal definition of a shell function is as follows:
`name () list ;`
- A function binds a *name* to the *list* of commands that composes the body of the function. The `(` and `)` characters are required at the function definition.
- The following examples illustrate valid and invalid function definitions:

```
lsl() { ls -l ; } # valid
```

```
lsl { ls -l ; } # invalid
```

- In this example, the first definition is valid but the second one is not because it omits the parentheses after the string `lsl`.

Creating and using functions II

- This example also demonstrates a common use of functions. Because the original shell, sh, did not have the alias keyword common to more recent shells, all aliases were defined in terms of shell functions.
- A frequently encountered example of this is the source command. The sh equivalent is the . command.
- Many converts from csh use the following function to simulate the source command:

```
source() { . "$@" ; }
```

- As this example shows, shell functions have a separate set of arguments than the shell script to which they belong.
- An important feature of shell functions is that you can use them to replace binaries or shell built-ins of the same name.

Creating and using functions III

- An example of this is:

```
cd () { chdir ${1:-$HOME} ; PS1="\`pwd\`$ " ;  
export PS1 ; }
```

This function replaces the `cd` command with a function which changes directories but also sets the primary shell prompt, `$PS1`, to include the current directory.

- To invoke a function, only its name is required, thus typing:

```
$ ls1
```

on the command line executes the `ls1()` function, but typing:

```
$ ls1()
```

does not work because `sh` interprets this as a redefinition of the function by the name `ls1`.

Creating and using functions IV

- In most versions of the shell, typing `lsl()` results in a prompt similar to the following:
 >

Examples I

- A simple task that is well-suited to a function is listing the current value of your PATH, with each directory listed on a single line.
- The basic shell code is:

```
OLDIFS="$IFS"  
IFS=:  
for DIR in $PATH ; do echo $DIR ; done  
IFS="$OLDIFS"
```

Here you save the value of IFS in the variable OLDIFS and then set IFS to :.

Examples II

- Because IFS is the Internal Field Separator for the shell, you can use the for loop to cycle through the individual entries in PATH.
- When you are finished, restore the value of IFS.
- To wrap this up in a function, insert the function name and the brackets as follows:

```
lspath() {  
    OLDIFS="$IFS"  
    IFS=:  
    for DIR in $PATH ; do echo $DIR ; done  
    IFS="$OLDIFS"  
}
```

Examples III

- Now you can run the function as follows:

```
$ lspath
```

- One of the main uses of this function is to check whether a particular directory is in your PATH.
- For example, to check whether /usr/dt/bin is in my path, I can do the following:

```
$ lspath | grep "/usr/dt/bin"
```
- Although the shell ignores directories in the path that does not exist, consider the following function, which deals with the issue getting your PATH variable set correctly:

Examples IV

```
SetPath() {  
    PATH=${PATH:= "/sbin:/bin"};  
    for _DIR in "$@"  
    do  
        if [ -d "$_DIR" ] ; then PATH="$PATH": "$_DIR" ; f  
    done  
    export PATH  
    unset _DIR  
}
```

- Here you set PATH to /sbin:/bin if it is unset.
- You can invoke this function as follows:

```
SetPath /sbin /usr/sbin /bin /usr/bin /usr/java/bin
```

Introduction

- Shell scripts are often called on to manipulate and reformat the output from commands that they execute.
- Sometimes this task is as simple as displaying only part of the output by filtering out certain lines. In most instances, the processing required is much more sophisticated.
- You will look at several commands that are used heavily as text filters in shell scripts.

The head command

- The basic syntax for the head command is

head [-n lines] files

Here *files* is the list of the files you want the head command to process. Without the -n lines option, the head command shows the first 10 lines of its standard input. This option shows the specified number of lines instead.

- Its real power happens in daily applications. Consider the need to generate a list of the five most recently accessed files in a public HTML files directory.
- To retrieve a list of the five most recently accessed files:

```
$ ls -1ut /home/rruiz/public_html | head -5
```


The tail command I

- The basic syntax for the tail command is similar to that of the head command:

tail [-n lines] files

Here *files* is the list of the files the tail command should process. Without the -n lines option, the tail command shows the last 10 lines of its standard input. With this option it shows the specified number of lines instead.

- Now consider the problem of generating a list of the five oldest mail spools on the system.

```
$ ls -lrt /var/spool/mail | tail -5
```

The tail command II

- An extremely useful feature of the tail command is the -f (f as in follow) option:

tail -f file

Specifying the -f option enables you to examine the specified file while programs are writing to it.

- If you have to look at the log files generated by programs that you are debugging, but you don't want to wait for the program to finish, you can start the program and then use tail -f for the log file.
- Some Web administrators use a command such as the following to watch the HTTP requests made for their system:

```
$ tail -f /var/log/httpd/access_log
```

Introduction I

- The `grep` command lets you locate the lines in a file that contain a particular word or a phrase.
- The word `grep` stands for *globally regular expression print*.
- The command is derived from a feature of the original UNIX text editor, **ed**. To find a word in `ed`, the following command was used:

`g/word/p`

Here *word* is a regular expression.

- The basic syntax of the `grep` command is:

grep word files

Here *files* is the name of a file(s) in which you want to search for *word*.

Introduction II

- The `grep` command displays every line in file that contains word. When you specify more than one file, `grep` precedes each of the output lines with the name of the file that contains that line.

Some grep features I

- The following command locates all the occurrences of the word *pipe* in file *chapter15.tex*

```
$ grep pipe chapter15.tex
```

- If grep cannot find a line in any of the specified files that contains the requested word, no output is produced.
- One of the features of grep is that it matches the specified word according to the case that you specify.
- Sometimes you want to match words regardless of the case that you specify. To do this, use the `-i` option.
- When no files are specified, grep looks for matches on the lines that are entered on STDIN. This makes it perfect for attaching to pipes. For example:

Some grep features II

```
$ who | grep rruiz
```

- Most of the time you use grep to search through a file looking for a particular word, but sometimes you want to acquire a list of all the lines that do not match a particular word.
- Using grep, this is simple: specify the -v option.
- For example, the following command produces a list of all the lines in /etc/passwd that do not contain the word home:

```
$ grep -v home /etc/passwd
```

- As grep looks through a file for a given word, it keeps track of the line numbers that it has examined.

Some grep features III

- You can have grep list the line numbers along with the matching lines by specifying the `-n` option. With this option the output format is:

file:line number:line

Here *file* is the name of the file in which the match occurs, *line number* is the line number in the file on which the matching line occurs, and *line* is the complete line that contains the specified word.

- Sometimes you don't really care about the actual lines in a file that match a particular word. You want a list of all the files that contain that word.
- By using the `-l` option of the grep command, you reach this:

```
$ grep -l pipe *
```

Introduction I

- Counting words is an essential capability in shell scripts. There are many ways to do it, with the easiest being the `wc` command.
- Unfortunately, it displays only the number of characters, words, or lines. What about when you need to count the number of occurrences of word in a file?
- The **tr** command (tr for *transliterate*) changes all the characters in one set into characters in a second set. Sometimes it deletes sets of characters.
- The **sort** command sorts the lines in an input file. If you don't specify an input file, it sorts the lines given on STDIN.

Introduction II

- The **uniq** command (uniq for *unique*) prints all the unique lines in a file. If a line occurs multiple times, only one copy of the line is printed out. It can also list the number of times a particular line was duplicated.

The tr command I

- To count the number of occurrences of word in a file, first you need to eliminate all the punctuation and delimiters in the input file because the word “end.” and the word “end” are the same.
- You accomplish this task using the tr command. Its basic syntax is:

tr 'set1' 'set2'

Here **tr** takes all the characters in *set1* and transliterates them to the characters in *set2*

- Usually, the characters themselves are used, but the standard C language escape sequences also work.
- To get an accurate count, all the words should be separated by spaces, so you need to convert all tabs and newlines to spaces:

The `tr` command II

```
$ tr '!"?:;\[ \]{}() ,.\t\n' ' ' < file
```

- Here I specified *set2* as the space character because words separated by the characters in *set1* need to remain separate after the punctuation is removed.
- Notice that the characters `[` and `]` are given as `\[` and `\]`. These two characters have a special meaning in **tr** and need to be escaped using the backslash character in order to be handled correctly.
- The next step is to transliterate all capitalized versions of words to a lowercase version. To do this, you tell **tr** to change all the capital characters 'A-Z' into lowercase characters 'a-z' as follows:

The tr command III

```
$ tr '!"?:;[\]\{\}(),.\t\n' ' ' < file |  
tr 'A-Z' 'a-z'
```

- At this point, several of the lines have multiple spaces separating the words. You need to reduce or squeeze these multiple spaces into single spaces to avoid problems with counting.
- To do this, you need to use the `-s` (s as in *squeeze*) option to the `tr` command.
- The basic syntax is:

```
tr -s 'set1'
```

When `tr` encounters multiple consecutive occurrences of a character in *set1*, it replaces these with only one occurrence of the character.

The tr command IV

- Try the following:

```
$ echo "feed me" | tr -s 'e'
```

```
$ echo "Shell Programming" | tr -s 'lm'
```

- Now you can squeeze multiple spaces in the output into single spaces using the command:

```
$ tr '!"?":;\[ \]{}() ,.\t\n' ' ' < file |  
tr 'A-Z' 'a-z' | tr -s ' '
```

The sort command I

- To get a count of how many times each word is used, you need to sort the file using the **sort** command.
- In its simplest form, the **sort** command sorts each of its input lines. Thus you need to have only one word per line.
- You can do this changing all the spaces into new lines as follows:

```
$ tr '!?"":;[\ ]{}(),.\t\n' ' ' < file |  
tr 'A-Z' 'a-z' | tr -s ' ' | tr ' ' '\n'
```

Now you can sort the output, by adding the sort command:

The sort command II

```
$ tr '!"":;[\]\{\}(),.\t\n' ' ' < file |  
tr 'A-Z' 'a-z' | tr -s ' ' | tr ' ' '\n' |  
sort
```

The uniq command I

- At this point, you can eliminate all the repeated words by using the `-u` (u as in unique) option of the `sort` command.
- Because you need a count of the number of times a word is repeated, you should use the `uniq` command.
- By default, the **uniq** command discards all but one of the repeated lines.
- The `uniq` command produces a list of the `uniq` items in a file by comparing consecutive lines.
- To function properly, its input needs to be a sorted file.
- You need `uniq` to print not only a list of the unique words in the *file* but also the number of times a word occurs.
- You can do this by specifying the `-c` (c as in count) option to the `uniq` command:

The uniq command II

```
$ tr '!"":;\[ \]{}() ,. \t \n' ' ' < file |  
tr 'A-Z' 'a-z' | tr -s ' ' | tr ' ' '\n' |  
sort | uniq -c
```

Sorting numbers I

- At this point the output is sorted alphabetically. Although this output is useful, it is much easier to determine the most frequently used words if the list is sorted by the number of times a word occurs.
- To obtain such a list, you need **sort** to sort by numeric value instead of string comparison.
- It would also be nice if the largest number was printed first. By default, sort prints the largest number last.
- To satisfy both of these requirements, you specify the -n (n as in numeric) and -r (r as in reverse) options to the sort command:

Sorting numbers II

```
$ tr '!"":;\[\]\{\}(),.\t\n' ' ' < file |  
tr 'A-Z' 'a-z' | tr -s ' ' | tr ' ' '\n' |  
sort | uniq -c | sort -rn
```

- By piping the output to **head**, you can get an idea of what the ten most repeated words are:

```
$ tr '!"":;\[\]\{\}(),.\t\n' ' ' < file |  
tr 'A-Z' 'a-z' | tr -s ' ' | tr ' ' '\n' |  
sort | uniq -c | sort -rn | head
```

- You used the sort -rn command to sort the output by numbers because the numbers occurred in the first column instead of the second column.

Sorting numbers III

- If the numbers occurred in any other column, this would not be possible.
- The sort command constructs a **key** for each line in the file, and then it arranges these keys into sorted order. By default, the key spans the entire line.
- The -k option gives you the flexibility of telling sort where the key should begin and where it should end, in terms of columns.
- The number of columns in a line is the number of individual words on that line. For example, the following line contains three columns:

files 80 100

Sorting numbers IV

- The basic syntax of the `-k` option is:

sort -k start,end files

Here *start* is the starting column for the key, and *end* is the ending column for the key. The first column is 1, the second column is 2, and so on.

Using character classes with `tr`

- The `tr` command knows several character classes, and the punctuation class is one of them. Table 1 gives a complete list of the character class names.
- The way to invoke `tr` with one of these character classes is:
`tr '[:classname:]' 'set2'`

Here *classname* is the name of one of the classes given in Table 1, and *set2* is the set of characters you want the characters in *classname* to be transliterated to.

- For example, to get rid of punctuation and spaces, you use the `punct` and `space` classes:

Using character classes with tr II

```
$ tr '[:punct:]' ' ' < file |  
tr '[:space:]' ' ' | tr 'A-Z' 'a-z' |  
tr -s ' ' | tr ' ' '\n' | sort | uniq -c |  
sort -rn | head
```

- I could also have replaced 'A-Z' and 'a-z' with the *upper* and *lower* classes, but there is no real advantage to using the classes.
- In most cases the ranges are much more illustrative of your intentions.

Character classes

Class	Description
alnum	Letters and digits
alpha	Letters
blank	Horizontal whitespace
cntrl	Control characters
digit	Digits
graph	Printable characters, not including spaces
lower	Lowercase letters
print	Printable characters, including spaces
punct	Punctuation
space	Horizontal or vertical whitespace
upper	Uppercase letters
xdigit	Hexadecimal digits

Table 1 : Character classes understood by the TR command.