Shell Programming and Unix Utilities

- Creating Shell Scripts
- Variables, Arguments, and Expressions
- Shell I/O and Debugging
- Testing Conditions and Control Statements
  - test, if, case, while, until, for, break, continue
- Exit Status
- Command Substitution
- Regular Expressions
- Utilities
  - grep, wc, touch, awk, tr, sort, gtbl, groff, ghostview, cut, head, tail, sed, gnuplot

Shell Scripts

- Advantages of schell scripts.
  - Can very quickly setup a sequence of commands to avoid a repetitive task.
  - Can easily make several programs work together to meet a set of goals.
- Disadvantages of schell scripts.
  - Little support for programming in the large.
  - Shell scripts are much slower since they are interpreted.

What Shell to Use

- The csh (C) shell and its derivative (tcsh) are recommended for use at the command line.
- The sh (Bourne) shell and its derivatives (ksh, bash) are recommended for writing shell scripts.
  - All shell script examples in this course are given using the Bourne shell syntax.

Finding Information about Bourne Shell and Its Commands

- Type "man <command>" or "info <command>" where <command> is the Bourne shell command of interest.
- Type "man sh" and look in the manual page.
- Look up information in the text or use other texts.
- See me or the TA.

Creating a Shell Script

- General convention is to use a .sh extension (<*name*>.sh) for shell scripts.
  - It is useful to know it is a shell script.
  - Some editors will do syntax coloring of shell commands.
- Type a special comment in the first line of each shell script file to indicate the shell to be invoked:
  - #!/bin/sh
  - Makes your shell scripts portable on different systems.
- Type in comments after this to indicate the purpose of the shell script.
  - # This shell script executes a program with specific arguments.
- Change the permissions on the schell script so it can be executed by typing the name of the file.
  - chmod +x <*name*>.sh

Printing a Line to Standard Output

- Use the echo command to print a line to stdout.
- The echo command always inserts a newline character at the end unless the -n option is used.
  - form
    - echo [-n] <*zero or more values*>
  - examples
    - echo "Hello World!"
    - echo "Please enter a value."
- The echo command is useful when debugging shell scripts.

Implementation of Echo

linprog4:~whalley/test/src

```
static char *sccsid = "@(#)echo.c       4.1 (Berkeley) 10/1/80";
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
        register int i, nflg;

        nflg = 0;
        if(argc > 1 && argv[1][0] == '-' && argv[1][1] == 'n') {
                nflg++;
                argc--;
                argv++;
        }
        for(i=1; i<argc; i++) {
                fputs(argv[i], stdout);
                if (i < argc-1)
                        putchar(' ');
        }
        if(nflg == 0)
                putchar('\n');
        exit(0);
}
                                                1,1           All
```


Shell Variables

- Shell variables are just used and not declared.
- The general convention is to use lowercase letters for shell variable names and uppercase letters for environment variable names.
- Shell variables are assigned strings, which can be interpreted as numeric values. Note that there can be no blanks before or after the '='.
  - form
    - <*name*>=<*value*>
  - examples
    - var=0
    - var="Good"

Shell Variables (cont.)

- Shell variables are initialized to be the empty string by default.
- Shell variables can be dereferenced.
  - form
    - $<*name*>
  - examples
    - var1=$var2
    - echo "====" $testcase "===="
    - echo "deleted" $num "lines from file" $file
- The number of characters of a shell variable in can be determined by using the form ${#<*name*>}.
- example
  - v="Hello World!"
  - echo "Width of v is" ${#v}"."
  - # Will cause the following line to be printed.
  - Width of v is 12.

Reading Values into a Shell Variable

- Use the *read* statement to read a line of standard input, split the line into fields of one or more strings that were separated by blanks or tabs, and assign these strings to shell variables.
- All leftover strings in the line are assigned to the last variable.
  - form
    - read <*var1*> <*var2*> ... <*varm*>
  - examples
    - read num
    - read name field1 field2
    - read name1 name2 < input.txt

Shell Arguments

- Can pass arguments on the command line to a shell script, just like you can pass command line arguments to a program.
  - run.sh 5
- $1, $2, ..., $9 can be used to refer to up to nine command line arguments (argv[1] ... argv[9]).
- Shell command line arguments beyond nine can be referenced referenced using the form ${#}, such as ${10}.
- $0 contains the name of the script (argv[0]).

Example Using Shell Arguments

- Example shell script called *prompt.sh*.
  - #!/bin/sh
  - echo "Please enter fields for employee" $1 $2"."
- Example usage:
  - prompt.sh John Doe
- Example output:
  - Please enter fields for employee John Doe.

More on Command Line Arguments

- $# contains the number of command line arguments.
- $@ will be replaced with a string containing the command line arguments separated by spaces.
- Example shell script called echo.sh.
  - #!/bin/sh
  - echo "The" $# "arguments entered were:" $@
- Example usage:
  - echo.sh cat dog horse
- Example output:
  - The 3 arguments entered were: cat dog horse
- Can reset the command line arguments using a *set* command.
  - #!/bin/sh
  - set -- cat dog cow
  - echo "The" $# "arguments entered were:" $@
- Output of the modified shell script would be:
  - The 3 arguments entered were: cat dog cow

Debugging Shell Scripts

- Shell scripts are interpreted by the shell.
  - Syntax errors are not detected until that line is interpreted at run time.
  - There is no shell script debugger.
- However, one can invoke a shell script with command line flags to assist in debugging.
  - The -v flag indicates the verbose option and the shell prints the commands as they are read.
  - The -x flag causes the shell to print the interpolated commands after they are read.

Debugging Shell Scripts (cont.)

- Invoke your shell script as follows so that each line of the script is echoed as it is interpreted.
  - form
    - sh -xv *<scriptname>*
  - example usage
    - sh -xv echo.sh a b c
  - example output
    - #!/bin/sh
    - echo "The" $# "arguments entered were:" $@
    - + echo The 3 arguments entered were: a b c
    - The 3 arguments entered were: a b c
- One can also set these options in the shell script itself.
  - The "set -xv" command will turn on these options.
  - The "set +xv" command will turn off these options.
  - One can also modify the initial comment to turn on debugging options.
    - #!/bin/sh -xv

Testing Conditions

- Can test for various conditions. There are two general forms.
  - test *<condition>* or [*<condition>*]
- I think the latter is easier to read. Be sure to include a space before and after each of the brackets.
- To test the result of a command, just use the command, which will return an exit status.
- Can reverse a condition by putting '!' before it.
  - [! *<condition>*]
- A ':' command always returns true.

Testing File Attributes

- Test if a file exists and is readable.
  - [-r employees.txt]
  - [-r data.txt]
- Test if a file doesn't exist or is not writeable.
  - [! -w employees.txt]
- Test if a file exists and is executable.
  - [-x echo.sh]
- Test if a file exists and is not a directory.
  - [-f tmp.txt]
- Test if a file exists and is a directory.
  - [-d tmp]

Numeric Tests

- The following {-eq, -ne, -gt, -ge, -lt, -le} operations can be used for numeric tests.
  - [$1 -lt $2]
  - [$1 -gt 0]
  - [$1 -eq $#]

Exit Command

- The exit command causes the current shell script to terminate.
- There is an implicit exit at the end of each shell script.
- The status indicates the success of the script with generally zero indicating success.
- If the status is not given, then the script will exit with the status of the last command.
- examples
  - exit 0
  - exit 1

String Relational Operators

- The string relational operators are:
  - =, !=, >, >=, <, <=
- The (>, >=, <, <=) operators assume an ASCII ordering when performing comparisons.
- They have to be used with the *expr* command, which will be covered later.
- The backslash has to be placed before these operators so they are not confused with I/O redirection.

Testing Strings

- Can perform string comparisons. It is good practice to put the shell variable being tested inside double quotes ("$v" instead of $v).
  - ["$1" = "yes"]
- The following will give a syntax error when $1 is empty since:
  - [$1 != "yes"]
- becomes
  - [!= "yes"]
- Can check if a string is empty to avoid a syntax error.
  - [-z $1]
- Can also check if a string is nonempty.
  - [-n $1]

Quoting Rules

- 'xxx' disables all special characters in xxx.
- "xxx" disables all special characters in xxx except $, ‘, and \.
- \x disables the special meaning of the character x.

Quoting Examples

- animal="horse"
- echo '$animal'
  - prints: $animal
- echo "$animal"
  - prints: horse
- cost=2000
- echo 'cost: $cost'
  - prints: cost: $cost
- echo "cost: $cost"
  - prints: cost: 2000
- echo 'cost: \$cost'
  - prints: cost: $cost
- echo "cost: \$$cost"
  - prints: cost: $2000

Testing with Multiple Conditions

- Can check if multiple conditions are all met.
  - ["$1" = "yes"] && [-r $2.txt]
- Can also check if one of a set of conditions are met.
  - ["$2" = "no"] || [! -r $2.dat]

General If Statement

- The "if *condition*", initial "then", and "fi" are required.
- Can have zero or more "elif *condition*".
- The "else" is also optional.
- General form.

```
if condition
then
   one-or-more-commands
elif condition
then
   one-or-more-commands
...
else
   one-or-more-commands
fi
```


If Statement Examples

```
if [ ! -r $1 ]
then
   echo "tmp.txt is readable."
fi

if [ $1 = $2 ]
then
   echo $1 "is equal to" $2
else
   echo $1 "is not equal to" $2
fi
```


If Statement Examples (cont.)

```
if [ $var1 -lt $var2 ]
then
   echo $var1 "is less than" $var2
elif [ $var1 -gt $var2 ]
then
   echo $var1 "is greater than" $var2
else
   echo $var1 "is equal to" $var2
fi
```


Case Statement

- Compares *stringvalue* to each of the strings in the patterns.
- On the first match it does the corresponding commands.
- ;; indicates to jump to the statement after the esac.
- *) means the default case
- General form.

```
case stringvalue in
pattern1) one-or-more-commands ;;
pattern2) one-or-more-commands ;;
...
*) one-or-more-commands ;;
esac
```

Case Statement Example

```
echo "Please answer yes or no."
read answer
case $answer in
"yes"|"Yes") echo "We are pleased you agree."
             ...
             ;;
"no"|"No")   echo "We are sorry you disagree."
             ...
             ;;
*)           echo "Please enter \"yes\" or \"no\"."
esac
```


While Statement

- The commands in the loop are performed while the *condition* is true.
- General form.

```
while condition
do
   one-or-more-commands
done
```


While Statement Examples

```
# process commands until a stop is encountered
read cmd
while [ $cmd != "stop" ]
do
   ...
   read cmd
done

# process a loop forever
while :
do
   ...
done
```


Until Statement

- The commands in the loop are performed until the condition is true.
- General form.

```
until condition
do
   one-or-more-commands
done
```

- Until Statement Example

```
read cmd
until [ $cmd = "stop" ]
do
   ...
   read cmd
done
```

Shift

- The shift command removes an argument from the script file's argument list by shifting all the others over one ($1=$2; $2=$3; $3=$4; ...).
- The shift command also decrements the $# value by one.
- Example:

```
# print the command line arguments, two per line
while [ $# -gt 0 ]
do
   echo $1 $2
   shift
   if [ $# -gt 0 ]
   then
      shift
   fi
done
```


Exit Status

- Zero normally indicates success.
- Nonzero values normally indicate some type of failure.
- It is a good practice to end each shell script with an "exit 0" command if everything succeeded.

Exit Command Example

- Below could be a script to get a yes/no answer.

```
#!/bin/sh
echo "Please answer yes or no."
read answer
while :
do
   case $answer in
     "yes") exit 0 ;;
      "no") exit 1 ;;
         *) echo "Invalid response."
            echo "Please answer yes or no."
            read answer ;;
   esac
done
```


Testing the Exit Status

- All conditions tested in control statements can also be the exit status of commands.
- The condition below uses the exit status of the yes.sh script shown in the previous slide.

```
if yes.sh
then
   echo "Please enter filename:"
   ...
fi
```

Regular Expressions

- Many shell commands and Unix utilities use regular expressions.
- A regular expression is a description of a possible sequence of symbols (e.g. characters).

Regular Expression Operations

- Concatentation is implicit.
  - `ab` # 'a' followed by 'b'
  - `abc` # 'a' followed by 'b' followed by 'c'
- * indicates zero or more instances of the preceding regular expression.
  - `a*` # "", a, aa, aaa, ...
  - `a*b` # b, ab, aab, aaab, ...
- + indicates zero or more instances of the preceding regular expression.
  - `a+` # a, aa, aaa, ...
  - `a+b` # ab, aab, aaab, ...

Character Classes

- '.' indicates any single character except a newline
  - `a.b` # 'a' followed by any character followed by 'b'
- Use [...] to indicate one of a set of characters. The '-' operator within [] indicates a range. The '^' after the '[' means match anything not in the set.
  - `[abc]` # a, b, c
  - `[0-9]` # any decimal digit
  - `[a-z]` # any lowercase letter
  - `[A-Z]` # any uppercase letter
  - `[a-zA-Z]` # any letter
  - `[^0-9]` # any character other than a decimal digit
  - `[^\n]` # same as '.'

Anchors

- Anchors can be used to indicate that a pattern will only match when it is at the beginning or end of a line.
- ^ indicates the beginning of the line.
- $ indicates the beginning of the line.
- Examples of regular expressions with anchors:
  - `^echo` # "echo" at the beginning of the line
  - `[A-Za-z]+$` # a name at the end of the line
  - `^done$` # "done" on a line by itself

Alternation and Grouping

- Use the '|' character to choose between alternatives.
- Use parentheses for grouping.
  - `a|b` # a or b
  - `a*|b` # "", a, aa, aaa, ..., b
  - `(ab)*c` # c, abc, ababc, abababc, ...
  - `(a|b)*c` # any combination of a's or b's followed by c

Testing If a Variable Matches a Regular Expression

- Can test if a variable matches a regular expression.
- General form.

```
[[ $var =~ regexpr ]]
```

- Example.

```
# check if $1 is not an acronym
if [[ ! $1 =~ ^[A-Z]+$ ]]
```


Additional RE References

- This table shows which symbols can be used within some Unix utilities.

Symbol	ed	vi	sed	awk	grep	Action
.	•	•	•	•	•	Match any character but newline.
*	•	•	•	•	•	Match zero or more preceding.
^	•	•	•	•	•	Match beginning of line.
$	•	•	•	•	•	Match end of line.
\	•	•	•	•	•	Escape character following.
[ ]	•	•	•	•	•	Match one from a set.
{ }	•		•		•	Match a range of instances.
+				•		Match one or more preceding.
?				•		Match zero or one preceding.
|				•		Separate choices to match.
( )				•		Group expressions to match.

Grep

- *grep* searches for strings in files that match a regular expression and prints the lines that contain these matches to stdout.
- The *pattern* below is the regular expression.
- You can specify zero or more files.
- If no files are specified, then grep reads from standard input.
- The [...] below means that whatever is inside the brackets is optional.
- The command line options will be covered later.

```
grep [-i] [-w] [-c] [-v] pattern [files]
```

Grep Examples

```
grep [Ww]halley *.txt
   # Detect Whalley or whalley in .txt files.
grep interger *.c
   # Detect where "integer" is misspelled as "interger".
grep ^\begin{table} report.tex
   # Are there any tables in report.tex?
grep Whalley report.tex | grep David > tmp.out
   # Where did I put my last and first name on the
     same line in report.tex.  Place output in tmp.out.
```


Grep Options

- -i will make the search case insensitive.
- -c will cause the number of lines matched to be printed.
- -w will make the search look for entire words.
- -v will cause the lines that don't match to be output.
- -A *num* will print *num* lines after each line that was matched.
- -B *num* will print *num* lines before each line that was matched.
- -C *num* will print *num* lines before and after each line that was matched.

Grep Examples Using Options

```
grep -i whalley tmp.tr
   # Finds both Whalley and whalley.
grep -c "do " prog.c
   # Counts number of do-whiles in prog.c.
grep -w "int abs" *.c
   # Find implementation of abs function.
grep -wc if prog.c
   # Counts if statements in prog.c.
grep -v "^#" prog.c
   # Prints preprocessor commands in prog.c.
grep -c "/"[*/] prog.c
   # Counts number of comments in prog.c.
grep -C 1 Whalley
   # Prints 3 lines around each line containing Whalley.
```


Wc

- *wc* counts the number of lines, words, and characters in files and prints this information to stdout.
- You can specify zero or more files.
- Like *grep* and most Unix utilities, it reads from standard input if no files are specified.
- Again the [...] below means that the ... inside the brackets is optional.
- The options indicate to only print the number of lines (-l), words (-w), or characters (-c). By default, it prints all three.

  `wc [-l] [-w] [-c] [`*files*`]`

Wc Examples

```
wc *.txt *.c        # How big are all of my .txt and .c files?
wc -l *.c           # How many lines of code have I written?
wc -w report.tex    # How many words are in my report?
wc -c doc.pdf       # How many bytes are in this document?
```


Touch

- *touch* creates a file with no data if it does not exist. Sometimes you need to create an empty file.

```
touch employee.txt
```

- *touch* updates the last modification date/time to be the current date/time if the file does exist.
- This feature can be useful if you copied files into a directory and you need to have the *Makefile* recompile everything.

```
touch *.cpp
make
```


Command Substitution

- A pair of backquotes, `...`, does command substitution.
- This means that the standard output of the command within the backquotes is used as arguments to another command.

```
count=`wc -w < $1`
   # assigns to count the number of words in file $1
if [ `wc -l < $2.txt` -lt 1000 ]
   # checks if number of lines in $2.txt file is < 1000
```


Xargs

- The *xargs* command reads a group of arguments from standard input and then runs the specified Unix command with that group of arguments.
- General form.

```
xargs <command>
```

- Example:

```
ls *.c > all_c_files.txt
   # capture *.c filenames
vi all_c_files.txt
   # can edit and delete some filenames
xargs gcc -g -c < all_c_files.txt
   # compile the list of files
```

For Statement

- The shell variable is assigned each word in the list, where the set of commands is performed each time the word is assigned to the variable.
- If the "in <word_list>" is omitted, then the variable is assigned each of the command-line arguments.
- General form.

```
for <variable> [ in <word_list> ]
do
   one-or-more-commands
done
```


For Statement Examples

```
# make a backup of each of the C files in the current directory
for file in `ls *.c`
do
   cp $file "$file".bak
done

# echo each of the command line arguments to a separate line
for arg
do
   echo $arg
done
```


For Statement Examples (cont.)

```
# place the command line args into a single string variable
# where the arguments are separated by blanks
s=
for arg
do
   s="$s $arg"
done
...
# compile each of the files specified in the list in $s
for file in $s
do
   gcc -g -c $file
done
```


Break and Continue

- A *break* statement causes immediate termination of a loop.
- A *continue* statement skips the rest of the commands in the loop and starts the next iteration.

```
for file in `ls *.c`
do
   if [ ! -r $file ] || [ -d $file ]
   then
      continue
   elif ! gcc -c $file
   then
      echo "could not compile" $file
      break
   fi
done
```

Expr

- Expr evaluates an arithmetic or relational expression and prints its results to standard output.
- Useful when your shell script needs to perform a calculation.
- Outputs 1 (true) or 0 (false) when evaluating a relational expression.
- Note that the arguments and the operators must be separated by spaces.
- At least one argument must be specified.

```
expr arg1 [ oper1 arg2 [ oper2 arg3 ... ] ]
```


Expr Examples

```
var=`expr "$var" + 1`
   # add 1 to var

if [ `expr "$s1" \< "$s2"` = 1 ]
   # check if $s1 < $s2, works for strings as well

a=`expr "$a" \* 2`
   # double the value of a
```