• I’m Dr. Ostermann, welcome!

• I have lots of bash stuff in my “prime” account
  – Just look in /home/osterman/
  – There are many shell scripts in /home/osterman/bin

• These slides are available in
  http://oucsace.cs.ohiou.edu/~osterman/talks/bash/
• Shell history and background

• Bash
  – Starting bash
  – Command-Line Editing
  – Command history

• Bash as a programming language
  – Shell variables
  – Wildcarding
  – Input/output redirection
  – Shell constructs

• Useful commands
  – Find
  – Sed
• The original Unix shell was written by Steve Bourne at AT&T in the early 1970s
  – Often installed as ‘‘bin/sh’’ and called the Bourne shell

• The next most common variant was “csh”
  – Written by a grad student at Berkeley to make the syntax more “C like” and added job control
  – Later variant was tcsh, which added more job control and fancy stuff
  – The csh should be avoided because of its broken syntax and poor semantics
The next variant was David Korn’s ksh, usually called the “Korn Shell” (also written at AT&T) in the early 1980s
- Added new constructs and more “builtin commands”
- If a script works in /bin/sh, it’ll work in /bin/ksh

The final common variant is the ‘Bourne-Again SHEll’, bash
- Commonly used by Linux users and hardcore programmers
- Uses the same syntax as sh/ksh
- Adds fancy command completion, command editing, etc...
- Available to the public under the gnu GPL
• The bash shell is commonly installed on “real” Unix machines as /bin/bash
  – That’s where you’ll find it on the “prime” machines too

• To start it from a different shell, just type “/bin/bash”

  p1% /bin/bash
  bash-2.05$
• It’s not possible to make bash your permanent shell on the “prime” machines
  – Mostly for security and sanity reasons

• Here’s what I do in my “.cshrc” file:

```bash
if ($?prompt) then
  #
  # well, we all know what I think about csh!!!!
  #
  exec /bin/bash
endif
```
• When bash is started as an “interactive shell” (but not a login shell), bash reads the commands in “∼/.bashrc”
  – This is where you can change environment variables (like PATH), write functions and aliases, etc, etc
  – See mine for a (big) example

• When started as an “interactive login shell”, bash reads the commands in “∼/.profile”
  – This is where you put things like terminal characteristics (how you backspace, stuff like that)
  – There shouldn’t be much in there
  – For simplicity, you will probably want to end it with
    . ~/.bashrc
• The most valuable thing that you can do to save yourself time is to learn real command line editing
  – It’s easy, and the bindings are useful in lots of other places

• The next most valuable thing is to learn emacs, but that’s another talk entirely!

• After that, comes gdb!
• Actually, this is the default in bash (as it should be)

• If you’re stuck in ksh, type “set -o emacs”
You’ll want to teach all of these commands to your fingers:

Moving
^a beginning of line
^e end of line
^f forward one character
^b backward one character
ESC f forward one word
ESC b backward one word

Editing
^d delete the current character
^h delete the previous character
^k kill the rest of the line

History
^p previous command
^n next command
^r previous matching command
Exercise 1
Command Line Editing

Type in the following commands and practice using the editing keys!

```
echo the quick brown fox jumped over the lazy dog
echo the quick RED fox jumped over the lazy dog
echo the quick red fox jumped over the lazy dog MORE THAN ONCE!
echo the quick red fox jumped
echo the quick fox jumped over the dog
```
• Since we’ve already talked about

  ^p previous command
  ^n next command
  ^r previous matching command

  ... we’ve already seen the bash history mechanism

• bash stores previous commands in the file
  “~/.bash_history”
  – You’ll probably want to make sure that this command isn’t world readable!
  – You can change the location with the environment variable HISTFILE

• You can change the maximum number of commands recorded in this file by changing the environment variable HISTSIZE (the default size is 500)
More History

- The command `history` will list all commands in your history
- The command `history N` will list the most recent N commands
- The command `history -c` will clear the command history
• The primary benefit of sh/ksh/bash over csh is that the former 3 interpreters also implement a *programming language*
  
  – csh does not

• Bash has
  
  – variables
  
  – looping constructs
  
  – functions
  
  – ...

Copyright 2002 - Shawn Ostermann
Shell Variables

- Legal characters in shell variable names
  - Upper and lower case letters
    - Beware, case is significant
      - By convention, we use all upper case!!
  - Digits
  - Underscore character

- All variables in the shell are of the same type: strings
Assigning Variables

• Assignments are of the form VARNAME=ONEWORD

VAR=1
FILENAME=./test.c
STR1=xyz
• To refer to a variable, prefix its name with a dollar sign
  
  – To avoid parsing ambiguity, it’s generally a good idea to surround the shell variable name with braces

```
STR=testing123
echo $STR
testing123
echo ${STR}
testing123
echo $STR456
echo ${STR}456
testing123456
```
Quoting and Parsing

- Before we get too far, we need to understand how the shell *tokenizes* (breaks into distinct words) the input.

- Tokens in the shell are divided by spaces, tabs, newlines and just about any character not legal in a variable name.

- Beware of

  ```bash
  VAR=testing ls
  ```
• All 3 sets of quotes have a different meaning
  – single (forward) quotes
    'this is a test: $VAR'
  • Group all the words together, but perform no expansion (no variables, wildcards, etc)
  – double quotes
    "this is a test: $VAR"
  • Group all the words together, **AND** perform expansion
  – back quotes
    ‘ls’
  • Run the command in the quotes and return its value (standard output) as a single word
The variables of the form $\text{DIGIT}$ refer to the script’s first 9 positional parameters

- $1$

Other useful ones

- ${\text{NUMBER}}$ for higher positional parameters (as in ${13}$)
- $#$ the number of positional parameters
- $0$ the name of the script
- * all of the arguments (separated as words, see the man page)
Other Well-Defined Shell Variables

- $HOME
  - The place that cd takes you back to (initialized to your home directory)
  - You can change this
  - It also affects what “~/” means

- $CDPATH
  - places to look for a directory when you say “cd dir” (very handy!)

- $PS1
  - Your prompt (see also $PS2-$PS4)

- $RANDOM
  - I’m not sure, it keeps changing! :-)

© Copyright 2002 - Shawn Ostermann
Environment Variables

- Shell variables are normally not visible to other programs
- Environment variables (or envariables) are a subset of the shell variables that are inherited by the programs that your shell calls
- If a variable is in the shell’s environment, it’s also a shell variable (and is referenced with the same syntax)
- To put a variable into the environment, use the syntax
  
  ```bash
  export VARNAME
  ```
  
  - This only needs to be done once, then it’s permanent until that shell exits
VAR="this is the variable VAR"
echo $VAR
this is the variable VAR
printenv VAR

export VAR
printenv VAR
this is the variable VAR
VAR="now it looks like this!!"
echo $VAR
now it looks like this!!
printenv VAR
now it looks like this!!
Common Environment Variables

- **PATH**
  - What you want other programs to start

- **SHELL**
  - Your userid (mine is "osterman")

- **HOME**

- **USER**
  - For X-Windows

- **DISPLAY**
  - What other programs should use to allow you to edit something (emacs of course!)

- **EDITOR**

- **TERM**
  - Terminal type (as in "vt100")
Pattern Substitution

- Other useful variable manipulation primitives
  
  ${name\%pattern}$
  - Remove pattern from the end of the value of name

  ${name\#pattern}$
  - Remove pattern from the beginning of the value of name

- Examples
  
  ```bash
  FILE=file.c
  echo ${FILE}
  file.c
  echo ${FILE%.c}
  file
  echo ${FILE#fil}
  e.c
  ```
Shell Wildcards

• The shell uses a simple syntax for expanding file names

• Common
  ?
  · Any single character
  · For example “ls file.?”
  *
  · any number of characters
  · For example “ls *.c”

• These can be grouped in very useful ways

  ls */*.c
  for DIR in */.; do echo $DIR; done
  rm -i */*xy*/file.?
  grep string */*/*

Other Useful Filename Patterns

- The tilde refers to a user’s home directory (or the HOME environment variable!)
  - The syntax “∼/.profile” is my bash startup file
  - The syntax “∼bob/.profile” is bob’s bash startup file

- The brackets allow you to allow (or deny) one of several options (always a single character)

  1. `ls *.ch`
  2. `ls *.!ch`
  3. `ls [a-z]*`
  4. `ls [!1-35-8]*`
By convention, every Unix program has access to three data streams

- Standard Input
  - Where it reads its input from
  - Normally the keyboard

- Standard Output
  - Where it sends its output
  - Normally the screen

- Standard Error
  - Where it writes error messages
  - Normally the screen
Redirecting Data Streams

- Standard input is redirected with `<
  - prog < ifile
  - vi < cmdfile

- Standard output is redirected with `>
  - program > output
  - ls > allfiles

- Standard error is redirected with `2 >
  - make 2> ERRS
The character `|` is a **PIPE**

- Not to be confused with bang (`!`)

Always separates two commands

- Makes the standard output of the first into the standard input of the second

  ```bash
  ls -R | more
  ```

Can be used many times

```bash
cat /usr/dict/words | grep compute | wc -l
```

Note that standard error is unaffected by the PIPE

```bash
prog 2> ERRS | grep -i "free money" 2> ERR2 | more 2>ERR3
```
• Something I call “piggybacking”

• The syntax “\texttt{N>>&M}” means “send file descriptor \texttt{N} to the same place as descriptor \texttt{M}”

• Examples

  \begin{verbatim}
  make 2>&1 | more
  make > alloutput 2>&1
  echo "fatal error" 1>&2
  \end{verbatim}
Quick Conditionals

• There are two conditional shorthands that are very useful
  – &&
    • continue if TRUE
    • make && ./run
  – ||
    • continue if FALSE
    • make || vi badfile.c

• Uses the exit value of the programs
  – zero is TRUE, non-zero is FALSE
Normal Conditionals

• Full example

```bash
if prog
  then
    lines1
  elif prog2
    lines2
  else
    lines3
fi
```

• as in

```bash
if true; then
  echo true
else
  echo false
fi
```
Useful Condition Testers
Test

- The `test` command goes back to the original `/bin/sh`
  - Normally lives in `/bin/test`
  - Built-in to most modern shells (for efficiency)
  - Has about 100 uses (type `man test`)
-e FILE
  • file FILE exists

-r FILE
  • file FILE exists and is readable

-w FILE
  • file FILE exists and is writable

-f FILE
  • file FILE exists and is a regular file

-d DIR
  • file FILE exists and is a directory
    (... and all of the other types too)

file1 -nt file2
  • file1 is newer than file2
string == string

string != string

-z "$VAR"

• is string length zero

-n "$VAR"

• is string length nonzero
Shell Constructs

Test
Other

- Numeric
  -eq -ne -ge -gt -le -lt
- Grouping
  -a AND
  -o OR
  -! NOT
  (expr) GROUPING
THEDIR=tmpdir
if test -e ${THEDIR} -a ! -d ${THEDIR}; then
    echo "Something is in the way"
    mv ${THEDIR} ${THEDIR}.SAVEME.${RANDOM}
else
    echo "Directory already exists"
fi

if test ! -d ${THEDIR}; then
    mkdir ${THEDIR}
fi
• In bash, the syntax `[[ test commands ]]` is a convenient, builtin shorthand
  – Almost the same as test (see the manual page)

• Example

  ```bash
  if [[ -n $DEBUG ]] ; then
    echo "Checking '$THISDISK' against '$UNWANTED'" 1>&2
  fi
  ```
Case Statements

- Commonly used for argument parsing

```bash
case $ARG in
  -d) DEBUG=true;;
  -h*) echo "Usage: $0 [-d] file [more files...]" 1>&2
       exit 1;;
  *) FILES="$FILES $ARG";;
esac
```

- Important notes
  - The right paren ends each pattern
  - The double semicolon is required at the end of each alternative
  - Only the first pattern found matches
  - Each of the patterns uses the wildcard syntax discussed above
• Definite iteration in which the loop variable is assigned each target in turn for one iteration of the loop

• General form:

```bash
for VARNAME in arg1 arg2 arg3
  do
    other scripts
  done
```

• For example

```bash
for DIR in */.; do
  ls ${DIR}
done
```
• Write a “one-liner” that will

  1) change the name of all of the “.c” files in the current directory to “.cc” instead

  2) move all the “.cc” files in the current directory to “.cc.old” instead
• Everybody should already know these utilities:
  – cat
  – rm
  – cp
  – tail
  – head
  – diff
- Cat stands for “concatenate” and is used to “stick a bunch of files together”.

```bash
cat file1 file2 file3
```
- Its “intended” use

```bash
cat file
```
- It’s short, I just want to see it

```bash
cat -v file
```
- Shows non-printing characters (NULL, control chars, etc)

```bash
ls | cat -v
```
- Find files with non-printing characters in them

```bash
cat > file
```
- And then just type - really quick editor!
Common Utilities

rm

• OK, it removes files, what’s the big deal?

  rm -f file
  – Try to delete a file even if the permission bits are wrong (also, don’t complain if it doesn’t exist)

  rm -i *
  – Prompt before removing each file (useful in scary situations!)

  rm -r *
  – Recursive (use sparingly, see -i)!

  rm -rf dir
  – If you’re REALLY sure (use carefully!)

  rm ./-f
  – Syntax is useful for removing files whose names make them look like flags!
cp -r dir /otherdir
   – Recursive copy

cp * /otherdir
   – Only the last argument needs to be a directory, everything else is moved there

cp -p file newfile
   – Preserves the creation and modification dates, owner, group, etc
tail file
  – List the last 10 lines of the file

tail -100 file
  – List the last 100 lines of the file

tail -f file
  – List the last 10 lines and then follow the file by listing
      new lines as they appear (great for growing output/log
      files)

tail -100f file
  – Start further back

tail +100 file
  – List the end of the file, starting 100 lines from the
      beginning of the file
head file

- List the first 10 lines of the file

head -100 file

- List the first 100 lines of the file

head *

- List the first 10 lines of all those files (nicely separated so you can tell what’s in them)

ls -lt | head

- Tell me a lot of info about the newest 10 files
diff file1 file2

- Are those files the same
  
  • There is no output if they’re identical

diff file1 thatdir

- Is file file1 the same as file thatdir/file1?

diff -r dir1 dir2

- Do dir1 and dir2 have all of the same files with the same contents?
• Usage

```bash
find files_and_directories pattern_definition
```

• Great for quick scripts and creating argument lists

• Is normally recursive

• Find is a “first generation” Unix command
  – Dates to early systems
  – Syntax is a little ugly
    • Backward from most other commands
find . -type f
  – Find all regular files recursively down from the current directory

find /bin /etc -type d
  – Find all directories starting at 2 places

find . -name '*.c'
  – Wildcard patterns (beware of quotes)

find . -mtime -2
  – Find all files with a modification time in the previous 2 days

find . -type l -o -type d
  – Find all symlinks and directories
    • Note that expressions are normally assumed to be separated by and (-a)
find . -size +1000c
   - Find all files larger than 1000 characters (default is blocks)

find . -type f -mtime -1 -size +1000000c
   - Find a recently-modified huge file

find /bin -type l -ls
   - Long listing of all symlinks in /bin

find /tmp -user sdo
   - Find all files belonging to me in /tmp

find . -type f -perm -111 -ls
   - Find all regular files with an execute bit turned on
The most useful expression is `-exec`

- Run a unix command
- The token `{}` is replaced with the name of the thing found
- The “command” ends with an escaped semicolon
  - Required even at the end of the line
find . -type d -exec chmod og-rwx {} \;
– Change the mode on all directories

find . -type f -mtime +365 -size -1000c -exec grep string {} \;
– Look for the string in all small files more than one year old

find /users/*/netscape/cache -type f -size +10000c -mtime +30 -exec rm {} \;
– Remove old, big netscape cache files

ls -lt ‘find /bin -atime +600‘
– Using find to create argument lists
• Write a “one-liner” to answer the following questions
  – Which programs in /bin are more than 1MB in size?
  – Are there any files in /tmp older than 1 month?
  – Find all files in your home directory (and below) accessed (atime) in the last 24 hours
  – Find all files that are empty
- **sed** is extremely useful for doing quick conversions of input files

- It’s pretty complicated, but it’s mostly just used these days for a single feature

- An example of simple substitution:

  ```bash
  echo "testing 1 2 3" | sed 's/ /_/g'
  ```

  will produce

  ```bash
  testing_1_2_3
  ```
• The pattern to match is a regular expression

• The pattern delimiters can be anything, but you’ll usually see the ‘/’ unless it’s needed for the pattern (this is slightly OS dependent)

• If the pattern is followed by a ‘g’ for global, then replace all matches on a line, the default is only to do the first
The other very useful feature of sed is the ability for a replacement to refer back to the pattern matched.

It’s extremely useful for swapping the order of columns in a file:

- \1 through \9 refer to the 1st - 9th matched regular expressions (surrounded by escaped parentheses)

A short example for reversing names:

```bash
#!/bin/ksh
sed 's/^([^a-zA-Z][a-zA-Z]*) ([^a-zA-Z][a-zA-Z]*)$/2, 1/'
```
• You’ll normally need single quotes around the substitution pattern

• Debugging complicated patterns is error prone
  – Start simple and add incrementally until it does what you want

• Most versions of sed only understand a subset of full regular expressions
  – No deeply-nested parenthesis
  – Can’t use ‘+’ (just use * and list the pattern twice)
• Write a “one-liner” that will
  1) change the name of all of the “.cc” files in the current
directory to “.c” instead
• I have lots of bash stuff in my “prime” account
  – Just look in /home/osterman/
  – There are many shell scripts in /home/osterman/bin

• These slides are available in
  http://oucsace.cs.ohiou.edu/~osterman/talks/bash/