input & output streams

- each UNIX program has access to three I/O "streams" when it runs:
  - **standard input** or **stdin**; defaults to the console keyboard
  - **standard output** or **stdout**; defaults to the console screen
  - **standard error** or **stderr**; defaults to the console screen
- the shell provides a mechanism for overriding this default behaviour (**stream redirection**)

stream redirection

- redirection allows you to:
  - take input from a file
  - save command output to a file
- redirecting from/to files using bash shell:
  - stdin:

```
% cmd < file
% less < ls.1
```

  - stdout:

```
% cmd > file                        # write
% ls -la >dir.listing
% cmd >> file                       # append
% ls -la /home >>dir.listing
```

  - stderr:

```
% cmd 2> file                       # write
% cmd 2>> file                      # append
```

stream redirection (2)

- redirecting stdin and stdout simultaneously

```
% cmd < infile > outfile
% sort < unsorted.data > sorted.data
```

- redirecting stdout and stderr simultaneously

```
% cmd >& file
% grep 'hello' program.c >& hello.txt
% cmd 1>out.log 2>err.log
```

- UNIX gotchas:
  - symbols used for redirection depend on shell you are using
  - our work will be with the Bash shell (`bash,sh`)
  - slight differences from C-shell's (`csh,tcsh`)

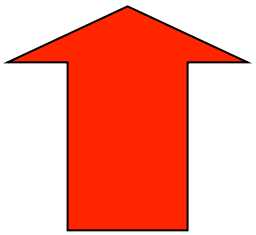
pipes

- Pipes are considered by many to be one of the major Unix-shell innovations
  - excellent tool for creating powerful commands from simpler components,
  - does so in an effective, efficient way.
- Pipes route standard output of one command into the standard input of another command
- Allows us to build complex commands using a set of simple commands
- Motivation:
  - without pipes, lots of temporary files result

without pipes

- Example: How many different users are currently running processes on the server?

```
% ps aux > temp1.txt
% awk '{ print $1 }' temp1.txt > temp2.txt
% sort temp2.txt > temp3.txt
% uniq temp3.txt > temp4.txt
% wc -l < temp4.txt > temp5.txt
% cat temp5.txt
```



Off by one – need to mentally subtract one from the resulting number

with pipes

- Example: How many different users are currently running processes on the server?

```
% ps aux | awk '{ print $1 }' | sort | uniq | wc -l
```

```
% ps aux | awk '{ print $1 }' | sort | uniq | wc -l | xargs expr -1 +
```

- Note the structure of the command:
  - "generator" command is at the head
  - successive "filter" commands transform the results
  - this is a very popular style of Unix usage

a bit more about pipes

- pipes can save time by eliminating the need for intermediate files
- pipes can be arbitrarily long and complex
- all commands are executed **concurrently**
- if any processing error occurs in a pipeline, the whole pipeline fails

command sequencing

- multiple commands can be executed sequentially, that is: `cmd1;cmd2;cmd3;…;cmdn`

```
% date; who; pwd
```

- may group sequenced commands together and redirect output

```
% (date; who; pwd) > logfile
```

- note that the last line does not have the same effect as:

```
% date; who; pwd > logfile
```

console

- UNIX needs to know your terminal type
- You can display what UNIX thinks it is with:
  ```
  % echo $TERM
  ```
- and re-set it to another terminal type with a command like
  ```
  % export TERM=vt100
  ```
- Several keyboard characters have special functions; the command
  ```
  % stty -a
  ```
  displays a list of special control characters.
- You can use stty to change these, e.g.,
  ```
  % stty erase ^X
  ```
  sets the erase (backspace) character on your keyboard to be ^X. (^X is entered by pressing X while holding down the **control** key).

Console (2)

- if a command expects input from standard input and no redirection is done, the program will take input from the keyboard
- a command that expects input on standard input may appear to "hang", typing ^D signals an end to input
- for example
  - command `wc` (when invoked without arguments) expects a text file from standard input
  - typing a string of characters followed by ^D supplies the input and signals the end of input.

Console (3)

- ^C will stop the current command
- XON/XOFF implements flow control to the console:
  - ^S (XOFF): halt output to the screen
  - ^Q (XON): restart screen output
- if you accidentally type ^S your terminal will freeze; restart output by typing ^Q
  - advice: try not to use explicit flow control; instead redirect or pipe program output using commands **less** or **more**
- Be careful with these
  - Stream speeds are now so fast that flow-control is often not the best way to control the terminal screen.
  - You may be better off using “less” or some other method of controlling output.

Introduction to UNIX (contd)

- Filename expansion
- Command aliases
- Quoting and backslash escapes
- **bash** command history
- Job control
- Shell/environment variables
- Customizing your shell

filename expansion

- "shorthand" for referencing multiple **existing** files on a command line

`*`	any number of characters
`?`	exactly one of any character
`[abc]`	any character in the set `[abc]`
`[!abc]`	any character **not** in the set `[abc]`

- these can be combined together as seen on the next slide

filename expansion (2)

- examples:
  - count lines in all .c files

    ```
    % wc -l *.c
    ```

  - list detailed information about all files with a single character file extension

    ```
    % ls -l *.?
    ```

  - send all Chap* and chap* files to the printer

    ```
    % lpr [Cc]hap*
    ```

filename expansion (3)

- * matches any sequence of characters (except those with an initial period)

```
% rm *.o   # remove all files ending in '.o'
% rm *     # remove all files in directory
% rm ../*-old*.c
```

- ? matches any single character (except an initial period)

```
% rm test.?       # remove test.c and test.o (etc.)
```

- So to delete a file of the form ".filename" you can't use wildcards

```
% rm .viminfo
```

How do we delete a file named *?

quoting

- controls bash's interpretation of certain characters
- what if you wanted to pass '>' as an argument to a command?
- **strong quotes** – All characters inside a pair of single quotes (') are preserved.
- **weak quotes** – Some characters ($,`) inside a pair of double quotes (") are expanded (interpreted) by the shell.
- **backquotes** – substitute result of evaluation as a command

quoting

```
% echo $SHELL *
/bin/bash file1 file2 file3

% echo '$SHELL' '*'
$SHELL *

% echo "$SHELL" "*"
/bin/bash *

%echo `date`
Thu Sep 17 14:59:34 PDT 2009
```

command aliases

- these allow a string to be substituted for a word when it is used as the first word of a command
- syntax

```
% alias mycmd='cmd [opt] [arg]'
```

- examples:

```
% alias more=less
% alias ls='ls -F‘
% alias rm=‘rm -i'
```

- to see a list of existing aliases (note no arguments):

```
% alias
```

- to undo / remove an alias

```
% unalias mycmd
```

backslash escaping

- Characters used by **bash** which may need to be escaped:

  ~, `, #, $, &, *, (,), \, [,], {, }, ;, ', ", <, >, /, ?, !

- single characters can be "protected" from expansion by prefixing with a backslash ("\")

  `cmd \*` is the same as typing `cmd '*'`

- protecting special characters in such a manner is an example of **backslash escaping**

```
% cp ~bob/junk \* # make copy of junk named '*'
% rm '*'        # remove '*' (not "delete all files")
```

- Single quotes around a string turn off the special meanings of most characters

```
% rm 'dead letter'
% cp ~bob/junk '*' # same as up above
```