• 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/
A shell script is a program that uses the shell constructs to drive the logic of some algorithm

- Sometimes, the entire program consists of builtin shell commands
- More often, the shell calls other Unix utilities to do the work
• The first line of a “script” is very important in Unix
  − Must be of the form below
  − The entire file (even the first line) is fed through the program indicated

`#!/bin/cat
testing 1 2 3`

`#!/bin/bash
echo wow, this works!`
• For a shell script to be run, it must be *readable* and *executable*

```bash
BASH:picard2> /bin/ls -l scriptfile
-rw------- 1 sdo 695 35 Nov 19 20:12 scriptfile

chmod a+rx scriptfile

BASH:picard2> /bin/ls -l scriptfile
-rwxr-xr-x  1 sdo  695 35 Nov 19 20:12 scriptfile
```
Debugging Scripts

- We've already seen two *builtins*: `echo` and `set`
- With no arguments, `set` shows you all variables (more later)
  - Handy arguments:
    - `set -x`
      - prints commands and assignments when executed
    - `set -v`
      - print commands before running them
    - `set -n`
      - just check syntax, don’t run the commands
    - `set +x`
      - turn off that annoying behavior!
- Handy at the top of the script

```
#!/bin/bash -x
# this is a simple example of debugging
echo testing 123
set
```
The last thing we have to know before we can play is the concept of the environment variable **PATH**

**The PATH is a colon-separated list of the directories that are searched by the shell (EVERY shell) when looking for commands**

- Security rule **#1**:  
  "*thou shalt never have the current directory, ".", in thy PATH!"

- Security rule **#2**:  
  "*no, really, never, not ANYWHERE, even at the end!"

- Null directories mean the same thing (and should be avoided!):  
  - `/bin::/usr/bin`
  - `/bin:`
  - `:/usr/bin`
Therefore we’ll always tell our shell **exactly** where we want it to find our scripts:

```bash
./script
~/bin/script
```
• Write a simple script called “ls” (for List Signature) that prints your name using the echo command

• Don’t forget to tell the shell to use THIS program ls!!!

• Don’t forget a comment at the top so you know why you wrote it!

• Now try it with the -x option
• All good shell scripts have an exit value:

```bash
#!/bin/bash
echo this is a program that works
exit 0
```

• Tells other programs (shells and make) that they worked correctly!

• For errors, use `exit 1` unless you want to have many different error codes, then use different (non-zero) numbers to indicate failure
Exercise 2
Simple Variables

• Write a simple shell script called `printargs` that prints out each of its (first 3) arguments:

```bash
BSH:picard2> examples/printargs
The name of this script is examples/printargs
Argument 1 is ''
Argument 2 is ''
Argument 3 is ''

BSH:picard2> examples/printargs testing
The name of this script is examples/printargs
Argument 1 is 'testing'
Argument 2 is ''
Argument 3 is ''

BSH:picard2> examples/printargs testing "1 2 3"
The name of this script is examples/printargs
Argument 1 is 'testing'
Argument 2 is '1 2 3'
Argument 3 is ''
```
Exercise 3
For Loop

- Rewrite your `printargs` program to work in general using a for loop
  - Remember `*$!!`
for ARG in $*; do
    case $ARG in
        -d) DEBUG=true;;
        -h) echo "Usage: $0 [-d] disk [more disks...]" 1>&2
            exit 1;;
        *) DISKS="$DISKS $ARG";;
    esac
done

if [[ -n "$DEBUG" ]]; then
    echo "Checking "$THISDISK" against "$DISKS"" 1>&2
fi

test -n "$DEBUG" && echo "you said: $DISKS" 1>&2
While loops use the exit value of a program to control how long they loop

- They're not generally as useful for shell scripts as FOR loops

```bash
FILE=./stopme
while [[ ! -f "${FILE}" ]] ; do
    echo "File ${FILE} is still missing"
    sleep 1
done
echo "Yea, the file is there!!"
```
FILE=.problem.cc
while [[ ! make ]]; do
    vi $FILE
done
echo "Yea, it works now!!"

Another While Example
• Use the `$( )` syntax to do math

• All calculations done as integers only

```
VAR=2
echo $VAR
echo $((VAR + 1))
echo $((VAR + 10))
echo $((VAR * 1024))
echo $((VAR / 1024))
```
VAR=1
while [[ $VAR -lt "10" ]]; do
    echo $VAR;
    VAR=$(($VAR + 1));
done
Exercise 4
While Loop and Math

• Use two nested FOR loops and the arithmetic operator to print the output:

```
1 * 1 is 1
1 * 2 is 2
1 * 3 is 3
1 * 4 is 4
1 * 5 is 5
2 * 1 is 2
2 * 2 is 4
[...]
4 * 5 is 20
5 * 1 is 5
5 * 2 is 10
5 * 3 is 15
5 * 4 is 20
5 * 5 is 25
```
• All of the following examples begin with:

    #!/bin/bash
    V0=""
    V1="x"
    V2="xx"
    V3="xxx"
    V4="xxxx"
    V5="xxxxx"
    V6="xxxxxx"
    V7="xxxxxxx"
    V8="xxxxxxxx"
    V9="xxxxxxxxx"
This is easy, it just uses string concatenation

```bash
echo "addition table"
for TERM1 in $V0 $V1 $V2 $V3; do
  for TERM2 in $V0 $V1 $V2 $V3; do
    echo "'${TERM1}' + '${TERM2}' = '${TERM1}${TERM2}'"
  done
done
```
addition table

'x' + 'x' = 'xx'
'x' + 'xx' = 'xxx'
'x' + 'xxx' = 'xxxx'
'xx' + 'x' = 'xxx'
'xx' + 'xx' = 'xxxx'
'xx' + 'xxx' = 'xxxxx'
'xxx' + 'x' = 'xxxx'
'xxx' + 'xx' = 'xxxxx'
'xxx' + 'xxx' = 'xxxxxx'
This is a little harder, but we need to make strings shorter

- Uses the syntax $\{\text{VAR}\%\text{PREFIX}\}$ to remove a suffix

```bash
echo "subtraction table"
for TERM1 in "$V3" "$V4" "$V5"; do
    for TERM2 in "$V0" "$V1" "$V2" "$V3"; do
        echo "'${TERM1}' - '${TERM2}' = '${TERM1%${TERM2}}'"
    done
done
```
subtraction table

'xxx' - '' = 'xxx'
'xxx' - 'x' = 'xx'
'xxx' - 'xx' = 'x'
'xxx' - 'xxx' = ''

'xxxx' - '' = 'xxxx'
'xxxx' - 'x' = 'xxx'
'xxxx' - 'xx' = 'xx'
'xxxx' - 'xxx' = 'x'

'xxxxx' - '' = 'xxxxx'
'xxxxx' - 'x' = 'xxxx'
'xxxxx' - 'xx' = 'xxx'
'xxxxx' - 'xxx' = 'xx'
echo "multiplication table"
for TERM1 in "$V1" "$V2" "$V4" "$V8"; do
  for TERM2 in "$V0" "$V2" "$V4" "$V8"; do
    echo -n "'${TERM1}' times '${TERM2}' = 
    SUM=""
    MULT=${TERM2}
    while [[ -n "$MULT" ]]; do
      SUM="${SUM}${TERM1}"
      MULT="${MULT#x}"
    done
    echo "'${SUM}"
    echo -n "${SUM}" | wc -c
  done
done

**Fun Math Examples**

**Multiplication - Output**

```
multiplication table
'x' times '' = ''
'x' times 'xx' = 'xx'
'x' times 'xxxx' = 'xxxx'
'x' times 'xxxxxxxx' = 'xxxxxxxx'
'xx' times '' = ''
'xx' times 'xx' = 'xxxx'
'xx' times 'xxxx' = 'xxxxxxx'
'xx' times 'xxxxxxxx' = 'xxxxxxxxxxxxxx'
'xxxx' times '' = ''
'xxxx' times 'xx' = 'xxxxxx'
'xxxx' times 'xxxx' = 'xxxxxxx'
'xxxx' times 'xxxxxxxx' = 'xxxxxxxxxxxxxx'
'xxxxxx' times '' = ''
'xxxxxx' times 'xx' = 'xxxxxxx'
'xxxxxx' times 'xxxx' = 'xxxxxxxxxxxxxxx'
'xxxxxx' times 'xxxxxxxx' = 'xxxxxxxxxxxxxxxxxxxxx'
'xxxxxxxx' times '' = ''
'xxxxxxxx' times 'xx' = 'xxxxxxxxxx'
'xxxxxxxx' times 'xxxx' = 'xxxxxxxxxxxxxxx'
'xxxxxxxx' times 'xxxxxxxx' = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
```

---

Math Examples

Copyright Shawn Ostermann 2005
echo "division table"
for NUMERATOR in "$V6" "$V7" "$V8" "$V9"; do
do
   for DENOMINATOR in "$V1" "$V2" "$V3" "$V4" "$V7"; do
do
      echo -n "'${NUMERATOR}' div '${DENOMINATOR}' = "
      COUNT=""
      REMAINDER=${NUMERATOR}
      while /bin/test ! "${REMAINDER}" '<' "${DENOMINATOR}" ; do
         REMAINDER="${REMAINDER}%${DENOMINATOR}""""""""
         COUNT="x$COUNT"
      done
      echo -n "'${COUNT}'"
      echo " with remainder ' ${REMAINDER}'"
   done
done
done
exit 0
'xxxxxx' div 'x' = 'xxxxxx' with remainder ''
'xxxxxx' div 'xx' = 'xxx' with remainder ''
'xxxxxx' div 'xx' = 'xx' with remainder ''
'xxxxxx' div 'xxx' = 'x' with remainder 'xx'
'xxxxxx' div 'xxxxx' = '' with remainder 'xxxxxx'
'xxxxxxx' div 'x' = 'xxxxxxx' with remainder ''
'xxxxxxx' div 'xx' = 'xxx' with remainder 'x'
'xxxxxxx' div 'xx' = 'xx' with remainder 'x'
'xxxxxxx' div 'xxx' = 'x' with remainder 'xxx'
'xxxxxxx' div 'xxxxx' = 'x' with remainder ''
'xxxxxxx' div 'x' = 'xxxxxxx' with remainder ''
'xxxxxxx' div 'xxx' = 'xx' with remainder 'x'
'xxxxxxx' div 'xxx' = 'xx' with remainder 'xx'
'xxxxxxx' div 'xxxxx' = 'x' with remainder ''
'xxxxxxx' div 'xxxx' = 'xx' with remainder 'x'
'xxxxxxx' div 'xxxx' = 'xx' with remainder 'xx'
Job Control

- Normally, the shell waits for each command to complete before giving a new prompt and waiting for a new command.

- If the command ends in an ampersand, it’s run in the background:
  - The shell issues a new prompt immediately.
  - The background process is disconnected from its standard input (assuming it’s the shell).
  - Standard output and standard error still go to the screen (unless they were redirected).

- Each background process is assigned a number by the shell.
• If you get tired of waiting for a command, you can force it into the background

BSH:OstermannTiBook> sleep 60
[ you type ^z ]
[1]+ Stopped sleep 60
BSH:OstermannTiBook> bg
[1]+ sleep 60 &
Job Control Commands

- jobs
  - List all background processes

- fg
  - Bring the most recently-used background process back to the foreground

- fg %N
  - Bring background process N back to the foreground
Simple Job Control Example

BSH:OstermannTiBook> sleep 60 &
[1] 4903
BSH:OstermannTiBook> jobs
[1]+ Running sleep 60 &
BSH:OstermannTiBook> sleep 60 &
[2] 4904
BSH:OstermannTiBook> sleep 60 &
[3] 4905
BSH:OstermannTiBook> jobs
[1] Running sleep 60 &
[2]- Running sleep 60 &
[3]+ Running sleep 60 &
BSH:OstermannTiBook> fg %1
sleep 60
You can terminate a background process using `kill`

```
BSH:OstermannTiBook> jobs
BSH:OstermannTiBook> sleep 30 &
[1] 4910
BSH:OstermannTiBook> jobs
[1]+  Running sleep 30 &
BSH:OstermannTiBook> kill %1
BSH:OstermannTiBook> 
[1]+  Terminated sleep 30
BSH:OstermannTiBook>
```
Still More Job Control

- The shell doesn’t normally want to let you log out if you have background processes.

- If you’re sure you don’t want the shell to think about them, use:

  (prog &)
  (xemacs file &)

- The parenthesis aren’t for grouping (although they do that too):
  - They spawn a sub-shell
  - That sub-shell makes a background process and then exits
  - That way the background process is unaffected if the original shell exits
  - Of course, you no longer have access to it with job control commands, either
    - You can still use the kill PID command
Other Useful Commands

- cron
- tar
- AWK
• Periodically, the cron daemon wakes up and checks the crontab file for each user

• If the timestamp fields match the current time/date, then the associated script is run
  – Userid is set to the owner of the time
  – All output is sent to the owner by email
A valid cron input file consists of lines of 6 fields

- Fields 1-5 specify the time to run
  - minute (0-59)
  - hour (0-23)
  - day of month (1-31)
  - month (1-12)
  - day of week (0(Sunday)-6(Saturday))

- The last field is the command to run at that time
  - Actually, anything past the 5th argument is passed to the shell
Cron Examples

0 03 * * *   echo I run at 3:00am every day
45 14 * * *   echo I run at 2:45pm every day
15 14 1 * *   echo I run at 2:15pm on the first of every month
30 06,14 * * * echo I run at 6:30am and 2:30pm every day
0 22 * * 1-5   echo I run at 10pm on weekdays
0,15,30,45 * * * echo I run every 15 minutes
5 4 * * 0      echo I run at 4:05am every sunday
Cron Recommendations

- Command should be a single shell script file name
  - Don’t do any redirection/etc in the crontab file
  - Much easier to debug!

- Cron shell scripts should specify the PATH explicitly at the top!
  - Otherwise you risk having the script behave differently when cron runs it
• Any standard output and standard error from the shell script is mailed to the owner
  - If there’s no output at all, then no email is sent

• Recommendation:
  - Only standard error should leave the shell script
  - Standard output should go into a log file
  - That way, if you get email, then something went wrong
• Back in the “good old days”, the system’s cron program only ran every 15 minutes
  – On my OSX workstation, it claims to run every minute

• Don’t trust it for anything with very tight timing bounds
  – Might run later than you want
• The “real copy” of your crontab file lives somewhere in the OS’s file space (usually /var/cron)

• The recommended interface is:
  – Type “crontab -l” to show your crontab file on stdout
  – Type “crontab -e” to edit your crontab file
    • Uses the environment variable VISUAL or EDITOR as the editor to use
  – Type “crontab newfile” to make newfile your crontab file
• I’ve always hated this interface, so I do it differently

• I leave my crontab file in my home directory
  – Lives in ~/.crontab

• One of the rules in my crontab is to update my crontab if it’s out of date!

• If you’re in a hurry, just run crontab .crontab
#!/bin/sh
#
# make sure my crontab file is read in correctly
#
PATH=/bin:/usr/bin
#
CRONTAB=$HOME/.crontab
SYSTEM_COPY=$HOME/.crontab.system
SYSTEM_SAVE=$HOME/.crontab.system.save
#
crontab -l > ${SYSTEM_COPY}
if cmp -s ${CRONTAB} ${SYSTEM_COPY}; then
  true
else
  date
echo "Your local .crontab file is out of date with respect to"
echo "the system copy. The differences were:"
echo diff ${CRONTAB} ${SYSTEM_COPY}
diff ${CRONTAB} ${SYSTEM_COPY}
    cp ${SYSTEM_COPY} ${SYSTEM_SAVE}
echo "The current system copy was saved as ${SYSTEM_SAVE}"
echo "I took the liberty of having your local file checked in."
crontab ${CRONTAB}
fi
# Shawn’s .crontab file
#
# minute (0-59),
# hour (0-23),
# day of the month (1-31),
# month of the year (1-12),
# day of the week (0-6 with 0=Sunday,1=Monday).
#
#min hr day mon wday event
#
# Nightly updates, every evening at 3am
00 03 * * * /home/sdo/etc/check_crontab
45 03 * * * /home/sdo/bin/change_dotplan
15 03 * * * /home/sdo/etc/check_motd
05 03 * * * /home/sdo/etc/check_mb_drafts
#
# Nightly updates, every evening at 4am
00 04 * * * /home/sdo/etc/nightly_update
#
# Nightly updates, every evening at 5am
45 05 * * * /home/sdo/etc/check_file_perms
#
# During class, make sure that the Ace www menus are up to date
30 06,14 * * * /home/sdo/etc/make_ace_menus # > /dev/null
# verify OAK email lists at 8am, noon, and 10pm
00 08,12,20 * * * /home/sdo/etc/check_oak_email
#
# copy local files to home daily (by 5:15pm)
#15 17 * * * /home/sdo/mac/incoming/update
#
# mrtg for email, every 15 minutes
0,15,30,45 * * * * /home/sdo/etc/mrtg_email/RUN
# backup mrtg for email, every night
15 4 * * * /home/sdo/etc/mrtg_email/COMMIT
#
#
#00 06,18 * * * /home/sdo/etc/backup_research
# turn off xhost every 15 minutes (in case I forget)
0,15,30,45 * * * * /home/sdo/etc/xhost_all_off
# class status reports every morning at 7am
0 7 * * * /home/sdo/etc/check_enrollments
# dump spamassassin’s spam into my inbox only at night (2am)
0 2 * * * /home/sdo/etc/inc_spam
• Create your own crontab file (and install it!)
  – It’s OK if you don’t have any rules in it
  – At least put the comments from the top of my example file at the top so you can remember what the fields are
• tar is a “first generation” Unix command and the syntax is a little strange
  – Because it predates modern Unix argument conventions, the arguments don’t need to be preceded with dash
    • If you see a tar command without dashes, it was likely written by an “old timer”

• In theory, tar is just the ar command with tape facilities built in
  – These days, it’s rarely used for actual magnetic tapes, but that’s what it was for
  – Normally just used to create archives
• Tar is designed to read/write from a tape drive by default
  – Uses /dev/rst0 by default
  – Since it’s unlikely that’s what you really want to do, you’ll always need to specify the “archive” file on disk
  – If the archive name is “-”, then it’ll use standard input or standard output (whichever is appropriate)
You can see the contents of a tar file using:

```bash
tar tf thefile
```
• Normally, you can just use
  \texttt{tar xf thefile}
  and \texttt{tar} will extract all of the files into the places where they should go

• I normally use
  \texttt{tar xvf thefile}
  which adds the \textit{verbose} flag which will show you the file names as it processes them
• If you only want some of the contents, you can list them
  tar xvf thefile thisfile thatfile thatdirectory
  and it will extract only the requested files (recursively if
  they’re directories)
  – You need to specify the prefix of the filename exactly or
    it won’t find it
  • Meaning directories prefixes, not just the file’s name
• The syntax for creating a tarfile is only a little more complicated

    tar cf tarfile .

will make a tarfile out of the current directory and recursively all of its children

• You need to at least specify one file/directory, but you can list several

    tar cf /tmp/tar /etc/passwd /etc/shadow /etc/rc.local
With older versions of tar, you need to be careful about the names that you use for the files

- If the file/directories in the tarfile were created with absolute path names, then the result when extracted will get installed relative to the root of the filesystem

- Most modern tar programs will refuse to install files with absolute pathnames
  - Obvious security implications
  - Strip off the leading slash
  - If you really want it to install relative to the root directory, then you need to “cd” there first
Beware!

• Beware of the following rather amusing “oops”

  tar cf tarfile *

if tarfile exists in the current directory when you run the command, then the tarfile gets included into the tarfile

  – Causes it to grow without bound!
  – Most modern versions of tar will detect this and complain
  • Sometimes by saying that one of the files is “growing”
Using tar To Copy Directories

Here’s a “one-liner” that I use often

```
tar cf - . | (cd /some/other/place; tar xvf -)
```

This makes an identical copy including all symlinks, permission info, timestamps, etc
• Normally tries to preserve owner, mode, and timestamp information
  – Not always possible for permission reasons
  – If you’re not running it as root (which you should probably never do!), then the files will be owned by “you”

• I always use “tar tf file” to look at the tarfile before extracting the contents so that I know where the files will go
  – Because it’s too late to change your mind after it starts if it overwrites something important
• Linux boxes and some other Unix varieties use the Gnu version of tar

• Like most Linux software, it’s badly bloated with excess arguments
  – Some of them are handy, but they’re not standard
  – Supports gzip’d files, for example
  – I try to avoid learning them because they’re not portable
• The AWK program dates back to 1977 at AT&T Bell Labs
  – Strange name comes from the initials of its authors
    • Alfred Aho
    • Peter Weinberger
    • Brian Kernighan
  – Recognize any of those names???

• AWK is best at processing a single input stream and doing a sophisticated analysis of its contents
• Awk is pretty well summed up by the following quote from the authors

Recipe For A Programming Language

1 part egrep   1 part snobol
2 parts ed    3 parts C

Blend all parts well using lex and yacc. Document minimally and release.

After eight years, add another part egrep and two more parts C. Document very well and release.
How AWK fits in

• We’ll see that we’ve discussed three “frameworks” for building a program
  – Shell Script
    • Good for miscellaneous tasks
  – make
    • Good for understanding timestamps and the existence of files
  – AWK
    • Good for understanding strings and patterns and input parsing
Different Versions

- There’s old AWK (1977)
  - You won’t find this one much

- There’s new AWK (1985)
  - Sometimes called nawk

- There’s GNU AWK (usually called gawk)
  - Found on Linux et. al.

- GAWK tends to have better error messages and is therefore easier to debug

- If you stick with the basics, nawk and gawk are interchangeable
The Basic Idea

- An awk program consists of a sequence of statements of the form:
  
  pattern { action }

- The input to awk (stdin or a file on the command line) is broken into lines.

- Each line is presented to each of the statements:
  - If the pattern matches, then do the action.

- If pattern is missing, it matches all lines.

- If action is missing, it means print the line.
• The actions look just like C for the most part
  – Variables are automatically converted between types to suit the operations applied to them
  – Variable references do **not** use the dollar sign
• $1, $2, ... refer to fields on the current line

• NR is the *record number*

• NF is the *number of fields*

• FS is a regular expression used to separate fields
  – You can also set it on the command line with `-Ffs`
• The pattern can be a regexp

```awk
/cat/ {print "I found a cat";}
```

• It can also use arbitrary expressions

```awk
NF > 10 {print "Lots of words " $0;}
length($0) > 72 {printf("Long line: %s\n", $0);}
```

• Two helpful static patterns
  
  − BEGIN
    
    • The action is called before the first line of the input
  
  − END
    
    • The action is called after the last line of the input
• The builtin command \texttt{next} means to skip the remaining patterns for the current input line
  
  – Recall that by default, awk checks \texttt{every} pattern

• You can also call builtin \texttt{exit}
Built-in Functions

- The set of built-in functions is rather limited
  - `length(s)`
  - `rand`
  - `substr(s, m, n)`
  - `index(s, t)`
  - `match(s, r)`
  - `sub(r, t, s)`
  - `gsub(r, t, s)`
  - `printf(fmt, expr, ...)`
  - `sprintf(fmt, expr, ...)`
  - `system(cmd)`
  - `tolower(str)`
  - `toupper(str)`

- `awk` has many more...
• Normally for shell scripts, the awk “program” is on the command line
  – Should almost always be surrounded by single quotes
  – Alternately, it can be a file given with awk -f progfile
#! /bin/ksh
#
# simple stats on the length of words in the dictionary file
#
awk ' 
    { len = length($1);
      ++count[len];
      if (len > max)
        max = len;
    }

    END { 
      for (i=1; i <= max; ++i)
        printf("Words of length %d: %d\n", i, count[i]);
    }
',

awk
```
#!/bin/ksh
PATH=/bin:/usr/bin:/usr/ucb
STRT=1
ADD=1
case $# in
  1) LIMIT=$1;;
  2) STRT=$1; LIMIT=$2;;
  3) STRT=$1; LIMIT=$2; ADD=$3;;
*) echo 'use: iota [start-number] limit-number [increment]' 1>&2
    exit;;
esac
# awk "BEGIN {for(i=$STRT;i<=$LIMIT;i+=$ADD) print i;exit 0}"
# exit 0
```
 associative arrays can be indexed by **any** value
  - Need not be an integer

• Makes many tasks much easier
#!/bin/sh
#
# Shawn Ostermann
#
# print the frequencies histogram of occurrences of the first word in each row of a file
#
# usage: freq [file]
#
# input format is assumed to be:
#   string <ignored>
#   string <ignored>
#   string <ignored>
#   ...

awk
awk 'BEGIN { histlen = 40;}
{ ++count[$1]; ++total; }
END {
 for (i in count) {
   if (count[i] > max)
     max = count[i];
   }
   for (i in count) {
     printf("%d\t%20s (%5.2f\%) \n", \n       count[i],i,100*count[i]/total);
     len = count[i]/max*histlen;
     for (j=1; j<=len; ++j)
       printf("*"),
     printf("\n");
   }
}' $* | sort -nr
Functions

- awk has simple functions
  - Can pass arguments
  - Can return results
- Note that all variables are global
awk

function graphit(total)
{
    for (i in count) {
        if (count[i] > max)
            max = count[i];
    }

    for (i in count) {
        printf("%d \t%20s (%5.2f\%)\n", \n            count[i], i, 100*count[i]/total);
        len = count[i]/max*histlen;
        for (j=1; j<=len; ++j)
            printf("*");
        printf("\n");
    }
}

BEGIN { histlen = 40; }
{ ++count[$1]; ++total;}
END { graphit(total);}
'
$* | sort -nr
• Parsing address book vcards is a prefect task for awk
  – Very regular format with fixed line prefixes

• Example vcards file

BEGIN:VCARD
VERSION:3.0
N:Hartman;Trick;;
FN:Trick Hartman
EMAIL;type=HOME;type=pref:phartman@ace.cs.ohiou.edu
NOTE:from MH aliases
END:VCARD
BEGIN:VCARD
VERSION:3.0
N:Hayes;Chris;;
FN:Chris Hayes
EMAIL;type=WORK;type=pref:chayes@cbsms1.cb.lucent.com
END:VCARD
I want the output to look like this:

```
# AUTOMATICALLY CREATED, DO NOT EDIT!!!!!!
#
# generated: Wed Dec 11 02:29:04 EST 2002
#
hartman_trick:   "Trick Hartman" <phartman@ace.cs.ohiou.edu>
hayes_chris:    "Chris Hayes" <chayes@cbsms1.cb.lucent.com>
```
awk -v DATE="'date''" 
function write_address(alias, firstname, lastname, tag, email_addr) { 
    fullname=firstname " " lastname; 
    sub(/^ */,"",fullname); 
    sub(/ *$/,"",fullname); 
    
    long_alias=sprintf("%s_%s", lastname, firstname); 
    
    gsub(/ /,"_",long_alias); 
    sub(/^_"","",long_alias); 
    sub(/ "$/,"",long_alias); 
    
    gsub(/ /*","",lastname); 
    gsub(/ /*","",firstname); 
    if (tag != "") { 
        long_alias=long_alias"_"tag; 
    } 
    long_alias=tolower(long_alias) 
    long_alias_colon=long_alias":" 
    
    if (alias != "") 
        printf("%s:\t\t\t\t%s\n", alias, long_alias); 
    
    printf("%-20s\t"%s" <\%s>\n", long_alias_colon, fullname, email_addr) 
}
BEGIN
    {FS="[:;]"
printf("# AUTOMATICALLY CREATED, DO NOT EDIT!!!!!!\n");
printf("#\n");
printf("# generated: %s\n", DATE);
printf("#\n");
}

/^BEGIN:VCARD$/ {alias=fullname=firstname=lastname=""; addr=0; delete email; ++ttl}

/^EMAIL;type=WORK;type=pref:/ {++addr; email["only"]=email["pref"]=email["work"]=

/^EMAIL;type=HOME;type=pref:/ {++addr; email["only"]=email["pref"]=email["home"]=

/^EMAIL;type=WORK:/ {++addr; email["only"]=email["work"]=$3; next}

/^EMAIL;type=HOME:/ {++addr; email["only"]=email["home"]=$3; next}

/^EMAIL;type=INTERNET/ {alias=$3;}

/^N:/ {firstname=$3; lastname=$2;}

awk

Other Useful Commands

Copyright Shawn Ostermann 2005
/^END:VCARD$/ { 
  if (addrs == 0) next;
  if (addrs == 1) {
    write_address(alias, firstname, lastname, "", email["only"])
  } else {
    for (TAG in email) {
      if (TAG=="only") continue
      if (TAG=="pref")
        write_address(alias, firstname, lastname, "", email[TAG])
      else
        write_address("", firstname, lastname, TAG, email[TAG])
    } 
  } 
}

END { 
  printf("Converted %d addresses\n", ttl) > "/dev/tty";
}

awk

Other Useful Commands

Copyright Shawn Ostermann 2005
Shell Scripting Hints

- A production shell script should always start by setting PATH
  - Makes them more useful for sharing with others because they don't depend on your local environment

- Never use csh for shell scripting!

- Always use exit values
#!/bin/bash
# print each of the arguments that I was given
echo "I was given $# arguments, as follows:"
for ARG in $*; do
  echo "  ${ARG}"
done
# exit 0