# Summer 2013 Handout 5

**Fix this with a temporary variable (Heisenberg Uncertainty Principle)**

Every time a shellscript runs a program, the program's exit status is put into the variable **$?**. See p. 140; **ksh**(1) p. 12. And every time you use **[[**square brackets**]]**, they put a **0** (for true) or a **1** (for false) into **$?**. See **ksh**(1) pp. 19−22.

```
#!/bin/ksh

prog

if [[ $? -eq 0 ]]
then
    echo prog was successful.
elif [[ $? -eq 1 ]]
then
    echo prog failed in the usual way.
else
    echo prog failed in an unusual way.
fi

exit 0
```

```
#!/bin/ksh

prog
s=$?

if [[ $s -eq 0 ]]
then
    echo prog was successful.
elif [[ $s -eq 1 ]]
then
    echo prog failed in the usual way.
else
    echo prog failed in an unusual way.
fi

exit 0
```

**Exit status of a pipeline**

```
prog1 | prog2 | prog3 | prog4
```

The exit status of a pipeline of programs is the exit status of the rightmost program (p. 145). This lets you say

```
if prog1 | prog2 | prog3 | prog4
then
    #programs to be executed if prog4 returns exit status 0
fi
```

For example,

```
#!/bin/ksh
#Output the login name of everyone in the class who works in area code
#212.  The work phone is on the 6th line of each bio file.

for filename in ~mm64/public_html/INFO1-CE9545/bio/*
do
    if awk 'NR == 6' $filename | grep 212 > /dev/null
    then
        head -1 $filename
    fi
done

exit 0
```

```
yb610
bc1478
jp3195
up244
```

**▼ Homework 5.1: search for Friday the thirteenth**

Write a shellscript named **fri13** that will output the numbers of the months and years from January 2013 to December 2017 inclusive that have a Friday the thirteenth:

```
1$ fri13
9 2013
12 2013
6 2014
2 2015
3 2015
11 2015
5 2016
1 2017
10 2017
```

Write two **while** loops (Handout 4, pp. 24, 29), or two **for** loops (**ksh**(1) p. 2.) Nest them as in Handout 4, pp. 9−10. The outer loop will be

```
while [[ $year -le 2017 ]]
do
```

Immediately after the word **if** in the inner **while** loop, pipe the output of **cal** into a **tail** that will remove the first two lines. (You get no credit if you remove only the first line. That means you can't say **tail +2**. Don't give **tail** a negative argument either.) Then pipe the output of **tail** into **awk** to remove everything except the column of Fridays:

```
2$ awk '{print $6}'                    the sixth field
3$ awk '{print $0}'                    the entire line
4$ awk '{print substr($0, 16, 2)}'     the 16th and 17th characters of the entire line, p. 117
5$ awk '{print substr($3, 16, 2)}'     the 16th and 17th characters of the 3rd field
```

You can eliminate the **tail** by having **awk** remove the first two lines of data. The **awk** should also remove every character except the sixteenth and seventeenth of each surviving line. (Or you can use the **cut** in Handout 4, p. 30, ¶ (2).) Then pipe the output of **awk** into a **grep > /dev/null**. If the **grep** finds what it's looking for, **echo** the variables that hold the month number and year number.

To make sure that 16 and 17 are correct for your version of **cal**, try this experiment before you write your shellscript. The experiment is not part of the shellscript.

```
6$ cal 1 2013 | tr ' ' .
...January.2013
.S..M.Tu..W.Th..F..S
.......1..2..3..4..5
.6..7..8..9.10.11.12
13.14.15.16.17.18.19
20.21.22.23.24.25.26
27.28.29.30.31
```

Better yet, have your shellscript output

```
7$ fri13
September 2013
December 2013
June 2014
February 2015
March 2015
November 2015
May 2016
January 2017
October 2017
```

by scalping the month name and year from the first line of the output of **cal**:

```
8$ cal 1 2013
   January 2013
 S  M Tu  W Th  F  S
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

You get no credit if any line of your output begins with a blank. Use **awk** to remove the indentation (Handout 4, p. 16):

```
9$ cal 1 2013 | awk 'NR == 1'
   January 2013

10$ cal 1 2013 | awk 'NR == 1 {print $1, $2}'
January 2013
```

You get no credit if you use **>>**, **$?**, or **`**back quotes**`**. You get no credit if you write **[[**square brackets**]]** after the word **if**. You get no credit if you use **$6** in the argument of **awk**: use **substr($0,** instead. You get no credit if you give the **-i** option to **grep**. You get no credit if you write a pipe

immediately before the word **if**:

```
prog | if

prog |
if
```

▲

**Exit status of a program with a pipe coming out of it**

```
#!/bin/ksh

prog1 | prog2 | prog3
```

Only the last program in a pipeline can have its exit status examined. To examine the exit status of **prog1** and **prog2** in the above example, we must run each program separately and use temporary files instead of pipes. See the **!** in Handout 4, p. 29.

```
#!/bin/ksh

if ! prog1 > ~/temp1
then
    exit 1
fi

if ! prog2 < ~/temp1 > ~/temp2
then
    exit 2
fi
rm ~/temp1

if ! prog3 < ~/temp2
then
    exit 3
fi
rm ~/temp2

exit 0
```

**Unix error messages are not redirected by > or |**

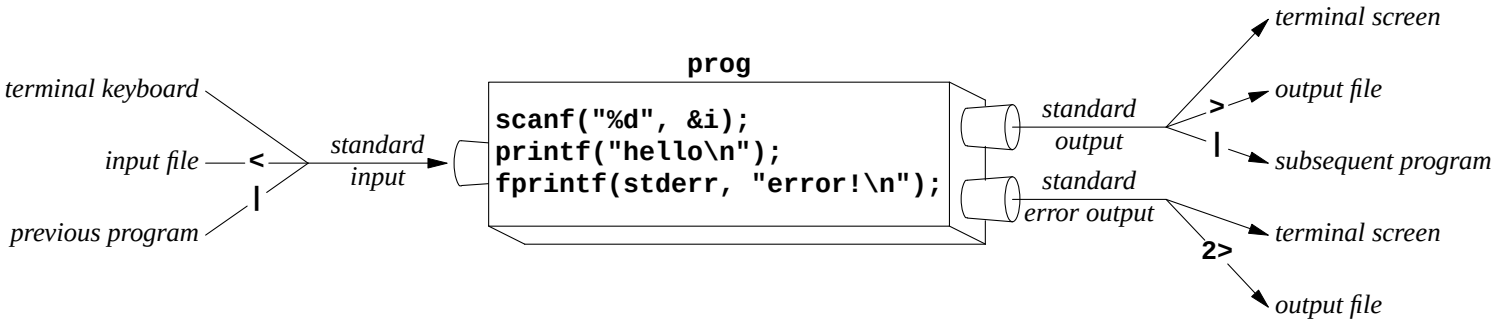
```
1$ cd /etc
2$ pwd

3$ grep 'root' passwd groups
passwd:root:x:0:0:root@i5:/root:/usr/bin/bash   standard output produced by grep
passwd:jh1997:x:16423:15:Jacqueline Harootian:/home1/j/jh1997:/usr/local/etc/expire
grep: can't open groups                          error message produced by grep

4$ grep 'root' passwd groups > outfile           Redirect the standard output to a file.
grep: can't open groups                          Error message was not redirected.
```

```
5$ grep 'root' passwd groups | wc -l
grep: can't open groups
       2
```

*Redirect the standard output to a program. Error message was not redirected.* **`wc -l`** *outputs this number.*

The diagram in Handout 2, p. 14, revealed only two-thirds of the truth about input and output:




Unfortunately, every Unix language uses a different trio of words for the source and two destinations:

| | *C file pointers* | *C++ streams* | *Perl filehandles* | *Ruby streams* | *Java streams* | *Korn shell file descriptors* |
|---|---|---|---|---|---|---|
| *standard input* | **`stdin`** | **`cin`** | **`STDIN`** | **`$stdin`** | **`System.in`** | **`0`** |
| *standard output* | **`stdout`** | **`cout`** | **`STDOUT`** | **`$stdout`** | **`System.out`** | **`1`** |
| *standard error output* | **`stderr`** | **`cerr`** | **`STDERR`** | **`$stderr`** | **`System.err`** | **`2`** |

The C file pointers are of data type **`FILE *`**. The C++ streams are of classes **`istream`** and **`ostream`**, which belong to namespace **`std`**. The **`c`** in **`cin`**, **`cout`**, **`cerr`** stands for ''character''. The Java streams are of classes **`InputStream`** and **`PrintStream`**, and are fields of class **`java.lang.System`**.

**How to write a Unix error message**

This shellscript sends its error messages to the screen even if its standard output is redirected into a file or a pipe. There must be no space embedded in the **`1>&2`**. See pp. 93, 141−142; **`ksh`**(1), pp. 22−23.

```
#!/bin/ksh

if [[ $# -ne 1 ]]
then
    echo $0: requires 1 command line argument 1>&2
    exit 1
fi

echo I received the argument $1.
exit 0
```

The name of the program and its command line arguments are counted by **`argc`** in C and C++. But only the command line arguments are counted by **`$#`** in the shell language, **`@ARGV`** in Perl, **`ARGV.length`** in Ruby, and **`argv.length`** in Java.

On a Unix platform, **`EXIT_SUCCESS`** is another name for the number 0, and **`EXIT_FAILURE`** for the number 1.

—On the Web at
**`http://i5.nyu.edu/~mm64/INFO1-CE9545/src/arg.c`**

```
1 #include <stdio.h>   /* C */
2
```

```
 3 int main(int argc, char **argv)
 4 {
 5     if (argc != 2) {
 6         fprintf(stderr, "%s: requires exactly 1 command line argument\n", argv[0]);
 7         return 1;
 8     }
 9
10     printf("I received the argument %s.\n", argv[1]);
11     return 0;
12 }
```

```
1$ gcc -o ~/bin/arg arg.c
```
*Minus lowercase O to create* **~/bin/arg**.

```
2$ ls -l ~/bin/arg
-rwx------   1 abc1234     users       7276 May 28 15:19 /home1/a/abc1234/bin/arg

    3$ arg hello
    I received the argument hello.

    4$ echo $?
    0
```
*See the program's exit status.*
*success*

—On the Web at
**http://i5.nyu.edu/~mm64/INFO1-CE9545/src/arg.C**

```
 1 #include <iostream>  //C++
 2 using namespace std;
 3
 4 int main(int argc, char **argv)
 5 {
 6     if (argc != 2) {
 7         cerr << argv[0] << ": requires exactly 1 command line argument\n";
 8         return 1;
 9     }
10
11     cout << "I received the argument " << argv[1] << ".\n";
12     return 0;
13 }
```

```
5$ g++ -o ~/bin/arg arg.C
```
*Create* **~/bin/arg**.

```
6$ ls -l ~/bin/arg
-rwx------   1 abc1234     users       8584 May 28 15:20 /home1/a/abc1234/bin/arg

    7$ arg hello
    I received the argument hello.

    8$ echo $?
    0
```

**http://i5.nyu.edu/~mm64/INFO1-CE9545/src/arg.pl**

```
#!/bin/perl

if (@ARGV != 1) {
    print STDERR "$0: requires exactly 1 command line argument\n";
    exit 1;
}

print "I received the argument $ARGV[0].\n";
exit 0;
```

**http://i5.nyu.edu/~mm64/INFO1-CE9545/src/arg.rb**

```
#!/home1/m/mm64/public_html/x52.9970/local/bin/ruby

if ARGV.length != 1
  $stderr.puts "#$0: requires exactly 1 command line argument"
  exit 1
end

puts "I received the argument #{ARGV[0]}."
exit 0
```

**http://i5.nyu.edu/~mm64/INFO1-CE9545/src/arg.php**

```
#!/usr/local/bin/php -q
<?php

if ($argc != 2) {
    $stderr = fopen('php://stderr', 'w');
    fwrite($stderr, "requires exactly one command line argument\n");
    fclose($stderr);
    exit(1);
}

echo "I received the argument " . $argv[1] . ".\n";
exit(0);
?>
```

—On the Web at
**http://i5.nyu.edu/~mm64/INFO1-CE9545/src/Arg.java**

```
 1 class Arg {
 2     static public void main(String[] argv) {
 3         if (argv.length != 1) {
 4             System.err.println("Arg: requires exactly 1 command line argument");
 5             System.exit(1);
 6         }
 7
 8         System.out.println("I received the argument " + argv[0] + ".");
 9         System.exit(0);
10     }
11 }
```

```
#!/bin/ksh
#Compile and run the Java program whose basename is the first argument.
#The remaining arguments, if any, are passed to the Java program itself.
#Sample use: jav Arg arg1 arg2 arg3

if [[ $# -ne 1 ]]
then
    echo $0: requires basename of Java program as argument 1>&2
    exit 1
fi

if javac $1.java
then
    java $*     #Handout 4, pp. 22, 24 for $*
fi
```

**Symbols for redirecting the error messages into a file**

There must be no space between the **2** and the **>**. See pp. 93−94; **ksh**(1) pp. 22−23.

*Send standard output into* **good_news**, *error messages into* **bad_news**:

```
1$ grep 'root' /etc/passwd /etc/groups > good_news 2> bad_news
```

*Send standard output and error messages into one big file:*

```
2$ grep 'root' /etc/passwd /etc/groups > all_the_news 2>&1
```

For example,

```
3$ gcc -o ~/bin/prog prog.c > prog.err 2>&1                    C
4$ g++ -o ~/bin/prog prog.C > prog.err 2>&1                    C++
```

**▼ Homework 5.2: trace the execution of a shellscript (not to be handed in)**

The command **set -x** causes a shellscript to print out each command line as it is executed, in addition to the normal output. See the textbook p. 136; **ksh**(1) pp. 46−50. Look up **set -e**, too.

```
#!/bin/ksh
#This shellscript is named shelly.
set -x

date
cal 5 2013
exit 0
```

```
1$ shelly
+ date
Tue May 28 15:20:02 EDT 2013
+ cal 5 2013
   May 2013
 S  M Tu  W Th  F  S
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

+ exit 0
```

The command lines are marked with a leading plus, stored in the variable **$PS4** (‘‘prompt string 4’’) in **ksh**(1) p. 17. Compare **$PS1** in Handout 2, p. 13, lines 39−42.

```
2$ echo $PS4
+
```

Put **set -x** into one of your shellscripts. Find out if the extra output belongs to the standard output or the standard error output. Then collect both outputs in one file.
▲

**The four flavors of uniq: pp. 106−107**

Duplicate lines must be adjacent in the input fed to **uniq**; **sort** is the easiest way to ensure this. We saw **uniq -d** in Handout 3, p. 15; **uniq** with no arguments Handout 3, p. 25; **uniq -c** in Handout 4, p. 18.

```
curly
larry
larry
larry
moe
moe
```

| uniq -d | uniq | uniq -c | uniq -u |
|---|---|---|---|
| larry | curly | 1 curly | curly |
| moe | larry | 3 larry | |
| | moe | 2 moe | |

**uniq -c examples: pp. 107−108**

The file **access_log** is poorly designed. The month and year are delimited by slashes and a colon, so it takes two **awk**’s to isolate them.

```
1$ cd /var/apache2/2.2/logs
2$ head -1 access_log
::1 - - [16/Jul/2012:16:08:02 -0400] "GET / HTTP/1.1" 200 44

3$ head -1 access_log | awk -F: '{print $1}'


4$ head -1 access_log | awk -F: '{print $1}' | awk -F/ '{print $2, $3}'
```

The longest line in the **access_log** is 1,475 characters long. If this causes **awk** to say "record too long", try the POSIX-compliant **/usr/xpg4/bin/awk** instead. If that **awk** also complains, use the following **perl** in place of the two **awk**'s. It looks for a word of three characters, a slash, and a number of four digits.

```
5$ head -1 access_log | perl -ne '/(\w{3})\/(\d{4}):/; print "$1 $2\n";'
Jul 2012
```

```
#!/bin/ksh
#How many web hits were there for each month?

perl -ne '/(\w{3})\/(\d{4}):/; print "$1 $2\n";' \
    /var/apache2/2.2/logs/access_log |
uniq -c   #unusual not to need sort before uniq
```

```
1684 Jul 2012
42760 Aug 2012
80457 Sep 2012
86510 Oct 2012
78487 Nov 2012
94723 Dec 2012
82900 Jan 2013
93853 Feb 2013
104269 Mar 2013
91091 Apr 2013
98390 May 2013
```

**▼ Homework 5.3: what is the busiest hour of the day?**

Write a shellscript that will produce the output in the last box in Handout 1, p. 8: a table showing how many hits there were during each hour of the day. If **awk** says "record too long", use the following **perl** instead.

```
1$ head -1 access_log | perl -ne '/:(\d{2}):/; print "$1\n";'
16
```

▲

If the following two **awk**'s say "record too long", use the following **perl** instead. **\s+** looks for white space; **\S+** looks for non-white space.

```
2$ head -1 access_log | perl -ne '/"\S+\s+(\S+)/; print "$1\n";'
/
```

```
#!/bin/ksh
#Output the names of the most frequently visited web pages,
#in decreasing order of how frequently they occur.
#Each name is preceded by the number of times it was downloaded

awk '-F"' '{print $2}' /var/apache2/2.2/logs/access_log |
awk '{print $2}' |
sort |        #put identical names on adjacent lines
uniq -c |     #count; only one copy of each name survives
sort -nr      #decreasing numeric order: "numeric reverse"

exit 0
```

```
82508 /
26077 *
22439 /favicon.ico
11469 /~mm64/planetarium/celx/
8795 /~mm64/planetarium/celx/moon.png
6968 /robots.txt
5760 /~mm64/planetarium/celx/orrery.jpg
5392 /~cmk380/cs101/USStates.txt
5139 /~cmk380/cs101/ProfessorSalary.txt
3728 /~mm64/INFO1-CE9236/
```

```
#!/bin/ksh
#Output the GID number of the group with the most members.
#Assume that each user belongs only to the group whose ID number
#is the fourth field of their line in the file /etc/passwd.

awk -F: '{print $4}' /etc/passwd |
sort -n |
uniq -c |
sort -nr |
awk 'NR == 1 {print $2}'

exit 0
```

```
15
```

**▼ Homework 5.4: a Christmas message**

The file **`$S45/christmas`** contains a hidden message of peace and goodwill for all mankind. Output a table in alphabetical order showing how many times each lowercase letter appears:

```
  57 a
  12 b
  69 c
  55 d
  77 e
```
*etc.*

(1) The command

```
tr -d '[a-z]\n'
```

(which you should not use in this homework) will read in lines of input, and output them with every lowercase letter and newline removed. **`-d`** stands for ''delete''.

(2) The command

```
tr -cd '[a-z]\n'
```

(which you should use in this homework) will read in lines of input, and output them with every character except the lowercase letters and newlines removed. **-c** stands for ''complement''. You will need a **<** to input a file into **tr**; see Handout 2, p. 23.

(3) The command

```
fold -b -1                                   minus one
```

will read in lines of input, and output them one character (''byte'') per line.

Use exactly one **sort**.

▲

**▼ Homework 5.5: who is doing nothing?**

Write a shellscript named **idlers** that will output the login name of everyone who is running exactly one process. Output nothing except the login names, one per line.

Let **ps -Af** provide the initial data, which will eventually be piped into the correct flavor of **uniq**. If you output the word **UID**, you forgot to do something and will get no credit. You will also get no credit if you use **who**, **w**, **finger**, or **ls**; or if you use **awk** more than once.

▲

**▼ Homework 5.6: how many different colors does X Windows have?**

> So vital was the exact degree of his seniority in a senator's career that elaborate—and rigid—formulas had been devised to determine it. Senators sworn in on the same day, for example, were ranked according to previous service in the Senate, followed by service in the House, and then within the Cabinet. If necessary, the holding of a governorship was factored in. And if it was still impossible to differentiate between two senators, . . . ''one may be declared senior to the other simply because his state was the earlier of the two involved to enter the Union.''
>
> —Robert A. Caro, *The Years of Lyndon Johnson: Master of the Senate*, p. 80

The file **/usr/openwin/lib/rgb.txt** lists the X Windows colors, one per line, with their red/green/blue components. The directory **X11** is uppercase X eleven.

```
1$ cd /usr/openwin/lib
2$ pwd

3$ ls -l rgb.txt

4$ cd
5$ pwd
```

```
6$ ls -l
total 648
drwxr-xr-x   3 mm64     users          3 Nov 11  2005 21
drwxr-xr-x   3 mm64     users          3 Nov 11  2005 23
drwxr-xr-x   5 mm64     users          5 Nov 11  2005 32
drwxr-xr-x   2 mm64     users          6 Nov 11  2005 40
-rw-------   1 mm64     users       6953 Apr  9  2012 42
drwxr-xr-x   4 mm64     users          4 Nov 11  2005 44
drwxr-xr-x   7 mm64     users          7 Jul 14  2009 45
drwxr-xr-x   6 mm64     users          6 Nov 11  2005 46
drwxr-xr-x   3 mm64     users          3 Nov 11  2005 47
drwxr-xr-x   3 mm64     users          3 Feb  8  2006 55
drwxr-xr-x   4 mm64     users          5 Oct  4  2007 64
drwxr-xr-x   3 mm64     users          3 Nov 11  2005 65
drwxr-xr-x   2 root     root          12 Feb 18  2007 TT_DB
-rwx------   1 mm64     users       9040 Mar 18 17:37 a.out
drwxr-xr-x   2 mm64     users         98 May 28 15:18 bin
-r--------   1 mm64     users       6322 Nov 14  2012 dave
-rwx------   1 mm64     users      10752 Nov 26  2012 echoserver
-rw-------   1 mm64     users       2068 Nov 26  2012 echoserver.C
drwxr-xr-x   2 mm64     users          4 May 25 11:55 gcc
-rw-------   1 mm64     users          3 May 17 18:01 j.txt
-r-x------   1 mm64     users         56 Mar  6 15:42 junk
-r--------   1 mm64     users        239 Mar 18 17:37 junk.C
-r--------   1 mm64     users       2968 Jul 13  2011 letter
-r--------   1 mm64     users       5914 Jan 12  2012 letter2
-r--------   1 mm64     users        674 Feb  3  2012 letter3
-r--------   1 mm64     users       2557 Feb  3  2012 letter4
-r--------   1 mm64     users       2926 Feb  3  2012 letter5
drwx------   2 mm64     users          5 Mar 15  2012 mail
drwxr-xr-x   3 mm64     users          3 Nov 11  2005 man
drwx------   6 mm64     users          6 Nov 11  2005 math
-rw-------   1 mm64     users       3048 May  8 14:38 named.ca
-r--r--r--   1 mm64     users         89 Apr 28 08:03 passwords
-r--r--r--   1 mm64     users         42 Apr 28 07:54 passwords2
drwx------   3 mm64     users         17 Mar 30 14:29 per
-r-xr-xr-x   1 mm64     users       1034 Jul 10  2012 post
drwxr-xr-x  72 mm64     users        202 May 28 14:50 public_html
-r--------   1 mm64     users        798 Nov 27  2012 reappointment
-rw-------   1 mm64     users       4665 Mar 19 09:58 selden
-r--r--r--   1 mm64     users      84049 Dec 16 08:57 std.celx
-rw-------   1 mm64     users      28006 Jan 16 10:14 string.h
-rw-------   1 mm64     users      90358 Jan 16 10:06 xutility
```

```
7$ head -6 rgb.txt
```

Verify that this file lists only 0 different colors even though it has lines, not counting the first. Output only one line, containing only one number—the number of different colors.

Feed the file to

```
8$ awk 'NR >= 2 {print $1, $2, $3}'
```

to remove the first line and the names of the colors and multiple blanks before piping it to **`sort`** and the correct flavor of **`uniq`**. How would **`awk`** confuse the following colors without the commas (p. 116)?

```
12 34 56                MurkyBlue
123 4 56                BrightRed
```

```
9$ awk 'NR >= 2 {print $1 $2 $3}'
```

You get no credit if you **sort** anything alphabetically anywhere in this shellscript. Use

```
10$ sort +0n +1n +2n
```

to sort in order of increasing redness, breaking ties by sorting in order of increasing greenness, breaking further ties by sorting in order of increasing blueness. See the **sort** in Handout 3, p. 28, Homework 3.11.

You get no credit if you use **cat** or **<** in this shellscript. Handout 2, p. 24, ★ shows how to avoid them.

▲

**▼ Homework 5.7: who are your ten biggest fans?**

Every computer connected to the Internet has an *IP address* made of four dot-separated numbers called *octets,* each in the range 0 to 255 inclusive. For example, the IP address of **i5.nyu.edu** is **128.122.109.53** (Handout 1, p. 19):

```
1$ /usr/sbin/nslookup i5.nyu.edu | awk 'NR == 5 {print $NF}'
128.122.109.53
```

Conversely, the fully qualified domain name of **128.122.109.53** is **i5.nyu.edu** or **i5.home.nyu.edu**. See the file **/etc/hosts**.

```
2$ /usr/sbin/nslookup 128.122.109.53 | awk '/name = / {print $NF}'
I5.HOME.NYU.EDU.
```

To see the IP address of each machine that has accessed one of your web pages,

```
3$ awk '/~abc1234/ {print $1}' /var/apache2/2.2/logs/access_log | more
```

and be patient. If **awk** says ''record too long'', try the POSIX-compliant **/usr/xpg4/bin/awk** instead. If that **awk** also complains, use

```
4$ perl -ane 'print "$F[0]\n" if /~abc1234/;' /var/apache2/2.2/logs/access_log
```

instead of **awk**. (The subscript in the **[**square brackets**]** is zero.) As you will see from the output of the above pipeline, there are some machines that have accessed your web pages many times.

Write a shellscript named **biggest_fans** that will output the IP addresses of the machines that are your ten biggest fans, i.e., the machines that accessed your web pages most frequently. List them one per line, in decreasing order of how many times each one has visited you. Precede each IP address with the number of times that machine has visited you. Prepare the IP addresses for **uniq** by passing them through the following pipeline, which we saw in Handout 3, p. 28.

```
5$ tr . ' ' | sort +0n +1n +2n +3n | tr ' ' .
```

Here are my ten biggest fans as of May 28, 2013.

```
12376 208.123.162.2
12266 192.168.66.96
7163 96.126.106.116
4126 96.242.183.10
4118 192.168.66.123
3868 68.193.205.150
3381 207.138.171.10
3261 192.168.66.95
2936 68.172.230.229
1854 24.193.121.52
```

▲

**Saturday Night Special**

Here's how I will prevent you from moving files into or out of the **~mm64/public_html/INFO1-CE9545/bio** directory. You'll have to put your file in the **~mm64/public_html/INFO1-CE9545/bb** directory instead, and then send me mail asking me to move it to the **~mm64/public_html/INFO1-CE9545/bio** directory. Do not put your bio file into any directory other than the **~mm64/public_html/INFO1-CE9545/bb** directory.

```
1$ cd ~mm64/public_html/INFO1-CE9545/bio
2$ pwd
/home1/m/mm64/public_html/INFO1-CE9545/bio

3$ ls -ld
drwxrwxrwx   2 mm64     users         10 May 28 15:15 .

4$ chmod 755 .                              Lock the files into the current directory.
5$ ls -ld
drwxr-xr-x   2 mm64     users         10 May 28 15:15 .
```

```
#!/bin/ksh
#This shellscript is ~mm64/bin/saturday.night.special,
#an interactive aid for moving all the bad biographies from
#~mm64/public_html/INFO1-CE9545/bio to
#~mm64/public_html/INFO1-CE9545/bb.

for filename in ~mm64/public_html/INFO1-CE9545/bio/*
do
    clear                    #Clear the screen and home the cursor.

    #Display the login name of the file's owner.
    #Then output an empty line.
    ls -l $filename | awk '{print $3}'
    echo

    #Display the required part of the biography.  Number the lines.
    head -12 $filename | cat -n
    echo

    echo 'Was it good or bad? \c'
    read verdict
    if [[ "$verdict" == bad ]]
    then
        echo Moving $filename to ~mm64/public_html/INFO1-CE9545/bb
        mv $filename ~mm64/public_html/INFO1-CE9545/bb
        sleep 3              #Give me time to read the message.
    fi
done

exit 0
```

**Is the standard output directed to a terminal? p. 151**

To make a shellscript give the **clear** command only when its standard output is going to a terminal, surround the **clear** with the following **if**. See **ksh**(1) p. 20. The **ls** program contains a similar **if**; see Handout 3, pp. 13−14.

```
if [[ -t 1 ]]
then
    clear
fi
```

**read: pp. 159−160**

The following **read** is not a program, so do not say **read**(1). Instead, see **ksh**(1) p. 45. For legibility, the prompt string should end with a blank.

```
#!/bin/ksh

echo 'Please type something and press RETURN: \c'
read something

echo You just typed $something.
exit 0
```

**Back quotes: print files in chronological order (pp. 86−88)**

***** is an abbreviation for an alphabetical list of the names of the files in the current directory. Therefore the command

```
1$ lpr *
```

would print these files in alphabetical order. To print them in chronological order,

```
2$ ls -t
brandnew
recent
fairlyold
bewhiskered

3$ lpr brandnew recent fairlyold bewhiskered
```

See **ksh**(1) pp. 5−6 for an easier way to print them in chronological order. **'**Single quotes**'** usually lean from upper right to lower left; **‘**back quotes**‘** lean from upper left to lower right. In some fonts, single quotes are thicker at the top; back quotes are thicker at the bottom.

```
4$ lpr  ‘ls -t‘                    Start with the newest: Handout 1, p. 10.
5$ lpr $(ls -t)

6$ lpr  ‘ls -tr‘                   Start with the oldest.
7$ lpr $(ls -tr)
```

How does the printout of line 4 differ from that of line 8?

```
8$ ls -t | lpr
```

**The two differences between a pipe and back quotes**

```
1$ prog1 | prog2
2$ prog2 ‘prog1‘
```

(1) When connected by a pipe, **prog1** and **prog2** run simultaneously. They start at the same time and end at the same time. See Handout 2, pp. 16−17. But when connected by back quotes, **prog2** does not start running until after **prog1** has finished.

(2) When connected by a pipe, the standard output of **prog1** becomes the standard input of **prog2**. But when connected by back quotes, the standard output of **prog1** becomes the command line arguments of **prog2**.

Many programs react differently to the same words depending on whether the words were received as standard input or as command line arguments. For example, **lpr** prints the words it receives as standard input. But it prints the contents of the files whose names it receives as command line arguments.

**Print every file except a.out**

**ls** will output the name of every file in the current directory. The pipeline

```
1$      ls | grep -v a.out          “invert”
```

will output the name of every file in the current directory except **a.out**. The command

```
2$ lpr ‘ls | grep -v a.out‘
```

will print (the contents of) every file in the current directory except **a.out**. How does this differ from

```
3$      ls | grep -v a.out | lpr
```

**Remove the five oldest files**

**`ls -tr`** will output the name of every file in the current directory in chronological order starting with the oldest. See Handout 1, p. 10 for **`-tr`**. The following pipeline will output the names of the five oldest files.

```
1$     ls -tr | head -5
```

The following command will remove them.

```
2$ rm ‘ls -tr | head -5‘
```

**Remove the five largest files**

The following command will output the names of the five largest files in the current directory.

```
1$     ls -l | tail +2 | sort +4nr | awk ’NR <= 5 {print $NF}’
```

The following command will remove them.

```
2$ rm ‘ls -l | tail +2 | sort +4nr | awk ’NR <= 5 {print $NF}’‘
```

**Remove the files but not the subdirectories**

Each of the two following commands will output the names of all the files in the current directory.

```
1$        ls -l | tail +2 | grep ’^-’ | awk ’{print $NF}’
2$        ls -l | awk ’NR >= 2 && /^-/ {print $NF}’    Handout 4, p. 17

3$ rm    ‘ls -l | awk ’NR >= 2 && /^-/ {print $NF}’‘   Remove all the files.
4$ rmdir ‘ls -l | awk ’NR >= 2 && /^d/ {print $NF}’‘   Remove all the subdirectories.
```

**Send mail to every delinquent**

Since **`delinquents`** and **`classmates`** (Handout 3, pp. 25−26) each output a list of login names, we can **`mail`** them **`letter`** with a single command:

```
1$ mail ‘delinquents‘ < letter
2$ mail ‘classmates‘ < letter
3$ mail ‘~mm64/bin/roster 45‘ < letter

4$ mail ‘who | awk ’{print $1}’‘ < letter
5$ mail ‘who | awk ’{print $1}’ | sort | uniq‘ < letter

6$ mail ‘awk -F: ’{print $1}’ /etc/passwd‘ < letter
7$ mail ‘awk -F: ’$4 == 15 {print $1}’ /etc/passwd‘ < letter
8$ mail ‘awk -F: ’$7 == "/bin/ksh" {print $1}’ /etc/passwd‘ < letter

9$ mail ‘ls -l ~mm64/INFO1-CE9545/homework | tail +2 | awk ’{print $3}’ | sort |
    uniq‘ < letter
```

To make the Saturday Night Special automatically send mail to the owner of each bad biography as it is moved to the directory **`~mm64/public_html/INFO1-CE9545/bb`**, insert the following immediately before the **`mv`**:

```
10$ mail ‘ls -l $filename | awk ’{print $3}’‘ < ~/letter
```

```
11$ finger ‘delinquents‘ | more
12$ finger ‘who | awk ’{print $1}’ | sort | uniq‘ | more
13$ finger ‘awk -F: ’{print $1}’ /etc/passwd‘ | more
14$ finger ‘awk -F: ’{print $1}’ /etc/passwd | sort‘ | lpr
```

**Make an index of everything in section 1 of the Unix manual**

```
1$ whatis cal

1. cal(1)      NAME          /usr/share/man/man1/cal.1
cal - display a calendar


2$ whatis cal grep

1. cal(1)      NAME          /usr/share/man/man1/cal.1
cal - display a calendar


1. grep(1)     NAME          /usr/share/man/man1/grep.1
grep - search a file for a pattern


2. pcregrep(1) NAME          /usr/share/man/man1/pcregrep.1
pcregrep - a grep with Perl-compatible regular expressions.


3. grep(1)     NAME          /usr/gnu/share/man/man1/grep.1
grep, egrep, fgrep - print lines matching a pattern


4. gegrep(1)   NAME          /usr/share/man/man1/gegrep.1
grep, egrep, fgrep - print lines matching a pattern


5. gfgrep(1)   NAME          /usr/share/man/man1/gfgrep.1
grep, egrep, fgrep - print lines matching a pattern


6. ggrep(1)    NAME          /usr/share/man/man1/ggrep.1
grep, egrep, fgrep - print lines matching a pattern
```

There are many other directories of documentation files; see your **$MANPATH** environment variable.

```
3$ cd /usr/share/man/cat1          the digit one
4$ pwd
5$ ls | sort -df | more            For -df, see Handout 1, p. 3, line 46.
accounts.ms
alias.ms
ancestry.ms
answer
arrow.ms
```

```
6$ ls | sed 's/\.[0-9]$//' | sort -df | more    Remove trailing dot digit; s for "substitute".
accounts.ms
alias.ms
ancestry.ms
answer
arrow.ms

7$ whatis `ls | sed 's/\.[0-9]$//' | sort -df` | more

1. pkgask(1m)  NAME          /usr/share/man/man1m/pkgask.1m
pkgask - stores answers to a request script


2. ckstr(1)    NAME          /usr/share/man/man1/ckstr.1
ckstr, errstr, helpstr, valstr - display a prompt; verify and return a
string answer


8$ whatis `ls | sed 's/\.[0-9]$//' | sort -df` | wc -l          Too long to print?
9$ whatis `ls | sed 's/\.[0-9]$//' | sort -df` | pr -l51 | lpr  minus L 51
```

Put the above command line 7 (without the **| more**) in a **while** or **for** loop to do the same for all nine sections of the manual. (They are in the subdirectories **cat2**, **cat3**, etc.)

**Ensure privacy for a different terminal each time: p. 68**

```
1$ tty
/dev/pts/10
```

Put the following **chmod** command in your **.profile** file to prevent another person from directing the output of their programs to your screen:

```
#mesg y                          Comment out this existing line.
chmod 600 `tty`                  Set your terminal to rw-------.
```

You'll have to say **exit** twice to log out:

```
2$ script script`date | tr ' ' -`
```

**Back quotes in a for loop**

```
1$ for loginname in `~mm64/bin/roster 45`      in place of Handout 4, p. 8
2$ for filename in file1 file2 file3
3$ for filename in *                           subdirectories as well as files
4$ for filename in `ls -l | awk 'NR >= 2 && /^-/ {print $NF}''     only the files
5$ for filename in `ls -lt | awk 'NR >= 2 && /^-/ {print $NF}''    start w/ newest
6$ for filename in `ls -ltr | awk 'NR >= 2 && /^-/ {print $NF}''   start w/ oldest

7$ for filename in `ls -l | awk 'NR >= 2 && /^-/ && $NF != "a.out" {print $NF}''
8$ for filename in `ls -l | awk 'NR >= 2 && /^-/ && $3 == "abc1234" {print $NF}''
9$ for filename in `ls -l | awk 'NR >= 2 && /^-/ && $5 >= 1000 {print $NF}''
10$ for filename in `ls -l | awk 'NR>=2 && $3=="abc1234" && $5>=1000 {print $NF}''
```

```
11$ for word in ‘cat ~/wordlist‘          ~/wordlist is a file containing words
12$ for word in ‘sort ~/wordlist‘
13$ for word in ‘grep itzky ~/wordlist | sort‘
```

**Print on the least busy printer**

```
#!/bin/ksh
#Print the files named as arguments.  Send them to whichever
#printer is less busy: edlab or ndlab.
#A more sophisticated version would add up the sizes of the files
#waiting to be printed.
#Sample use: fastest file1 file2 file3

if [[ ‘lpq -Pedlab | wc -l‘ -lt ‘lpq -Pndlab | wc -l‘ ]]
then
    echo printing $* on Ed Site edlab    #see p. 83 for $*
    lpr -Pedlab $*
else
    echo printing $* on North Dorm ndlab
    lpr -Pndlab $*
fi

exit 0
```

```
if [[ ‘who | wc -l‘ -gt 20 && ‘date | awk -F, ’{print $1}’‘ == Friday ]]
then
    echo Aw, give up: more than 20 people are logged in
    echo and today is Friday.
    exit
fi
```

**Add a test to the Saturday Night Special**

Why not use **-ne** instead of **!=**?

```
if [[ ‘ls -l $filename | awk ’{print $3}’‘ != ‘head -1 $filename‘ ]]
then
    echo The login name on line 1 is incorrect.
fi

if [[ "$verdict" == BAD || "$verdict" == Bad || "$verdict" == bad ]]
if [[ "‘echo $verdict | tr ’[A-Z]’ ’[a-z]’‘" == bad ]]
```

**Redirect a program's output to a local variable**

```
#!/bin/ksh

x=hello           #No space on either side of the =.
echo $x           #Verify that the assignment worked.

myname=‘whoami‘   #Redirect program’s output to a variable.
echo $myname      #It prints abc1234.

myname=whoami     #What happens if you forget the back quotes?
echo $myname      #It prints the word whoami.

exit 0
```

**Rename all the files**

```
#!/bin/ksh
#Change the names of all of the files in the current directory from
#uppercase to lowercase--for example, AUTOEXEC.BAT to autoexec.bat.
#Change only the files’ names, not their contents.
#Sample use: rename

echo ’Are you sure you want to rename all these files? \c’
read answer

if [[ "$answer" != yes ]]
then
    exit 1
fi

for filename in *
do
    echo $filename      #so I can watch the progress on the screen
    mv $filename ‘echo $filename | tr ’[A-Z]’ ’[a-z]’‘
done

exit 0
```

```
#!/bin/ksh
#Change the names of all of the files in the current directory from
#uppercase to lowercase, except when this would remove a file (e.g.,
#when there are two files named MOE and Moe).

status=0                #innocent until proven guilty

for filename in *
do
    newfilename=‘echo $filename | tr ’[A-Z]’ ’[a-z]’‘
    if [[ -e $newfilename ]]     #For -e, see p. 140; ksh(1) p. 16
    then
        echo $0: found $filename and $newfilename 1>&2
        status=1
    else
        echo $filename
        mv $filename $newfilename
    fi
done

exit $status
```

**What are the fully qualified domain names of your biggest fans (Homework 5.7)?**

```
1$ /usr/sbin/nslookup 128.122.109.53 | awk ’/name = / {print $NF}’
I5.HOME.NYU.EDU.
```

Unfortunately, our **nslookup** produces exit status 0 even if it couldn’t find a name for the host.

```
#!/bin/ksh
#Append hostname, if there is one, to each line output by biggest_fans.

biggest_fans |
while read line
do
     n=‘echo $line | awk ’{print $1}’‘
    ip=‘echo $line | awk ’{print $2}’‘

    echo $n $ip ’\c’
    /usr/sbin/nslookup $ip > ~/out 2> ~/err

    if grep "can’t find" ~/err > /dev/null
    then
        echo     #Output a newline.
    else
        awk ’/name = / {print $NF}’ ~/out
    fi

    rm ~/out ~/err
done

exit 0
```

If **nslookup** produced an exit status of 0 only for successful termination, we could avoid creating **~/err**:

```
if /usr/sbin/nslookup $ip > ~/out 2> /dev/null
then
    awk '/name = / {print $NF}' ~/out
else
    echo
fi
```

To identify hosts that have no names, use **whois** at **http://www.geektools.com/**.

```
12376  208.123.162.2   208-123-162-2.cust-nwp.nuvisions.net.
12266  192.168.66.96   ND-IMAC-19.NDLAB.ITS.NYU.EDU.
7163   96.126.106.116  li363-116.members.linode.com.
4126   96.242.183.10   static-96-242-183-10.nwrknj.fios.verizon.net.
4118   192.168.66.123  ND-IMAC-06.NDLAB.ITS.NYU.EDU.
3868   68.193.205.150  ool-44c1cd96.dyn.optonline.net.
3381   207.138.171.10  ryevpn.asg.com.
3261   192.168.66.95   ND-IMAC-03.NDLAB.ITS.NYU.EDU.
2936   68.172.230.229  cpe-68-172-230-229.nj.res.rr.com.
1854   24.193.121.52   cpe-24-193-121-52.nyc.res.rr.com.
```

**▼ Homework 5.8: Monday's child is fair of face**

Write a shellscript named **day** that will output a different reminder each day of the week. For example, on any Wednesday it will output

```
6:00 Unix class at 7 East 12th Street, room 228
```

If you don't need to be reminded of anything, output the relevant line in the file **$S45/monday**. You get no credit if you make the ☜ mistake in Handout 5, p. 22.

**day** should accept no command line arguments, so the first order of business is to output an error message and **exit** if there were any arguments. The rest of **day** will be a chain of seven **if-then-elif**'s using back quotes. Format and indent your chain of **if-then-elif-else-fi** statements in exactly the same way as the last box in Handout 4, p. 20; you get no credit if there is any difference at all. Add a final **else** just in case the **date** command outputs a bad word, which you must output as part of the error message. You get credit only if the error message includes the bad word. An error message must have the three trimmings on the error message in the shellscript in Handout 5, p. 5; you get no credit otherwise.

(1) The error message must begin with the name of the program (**$0**), a colon, and a blank.

(2) The error message must be directed to the standard error output (**1>&2**) not the standard output.

(3) The program must yield a non-zero exit status in case of error, 0 otherwise.

*Don't try to output apostrophes:* the shell will think they're single quotes and become confused. Don't forget Saturday and Sunday. When **day** works, put the command

```
day
```

into your **.profile** file.

```
1$ date
Tue May 28 15:22:29 EDT 2013

2$ date | awk -F, '{print $1}'
Tue May 28 15:22:29 EDT 2013
```

Use a variable to avoid having to run **date** and **awk** seven times:

```
day=‘date | awk -F, ’{print $1}’‘

if [[ $day == ...
then
    echo ...
elif [[ $day == ...
```

Another way to avoid multiple **date**'s and **awk**'s is by using a **case** statement instead of a chain of seven **if**'s. See pp. 134−135, **ksh**(1) p. 2.

You get no credit if your program requires a command line argument: it should figure out the day of the week all by itself. You get no credit if your program has more than two **fi**'s: use **elif** to eliminate the need for a large number of **fi**'s. Do not use **set**.

To test the seven messages, wrap a **for** loop around the chain of **if-then-elif**'s, but remove the **for** loop before you hand it in:

```
for day in Monday Tuesday Wednesday Thursday Friday Saturday Sunday Garbage
do
    #day=‘date | awk -F, ’{print $1}’‘
    if [[ $day == Monday ]]
    then
        ...
done
```

▲

**▼ Homework 5.9: don't hardwire the name of the directory into the shellscript**

We hardwired the directory name **~/public_html** into the shellscript **post** in Handout 4, p. 23, Homework 4.3. But it would be more portable for **post** to read this name from the **httpd.conf** file in Handout 3, pp. 7−8, using the **awk** command in Handout 4, p. 18, line 4. Verify that you have permission to read **/etc/apache2/2.2/httpd.conf**, and then

```
dirname=~/‘awk ’$1 == "UserDir" {print $2}’ /etc/apache2/2.2/httpd.conf‘
```

Then change every **~/public_html** in **post** to **$dirname**. Also add **1>&2** and a different exit status number for each **exit**.

▲

**How not to use back quotes**

Never write a pair of back quotes that are entirely occupied by one **echo**:

```
1$ cal ‘echo 5 2013‘    ☜          2$ cal 5 2013
3$ cal ‘echo $month $year‘         4$ cal $month $year
```

Similarly, never use one pair of back quotes to provide all the arguments for an echo:

```
5$ echo ‘date‘                     6$ date
7$ rm ‘ls *.o‘                     8$ rm *.o

9$ prog > ~/temp
10$ x=‘cat ~/temp‘
11$ rm ~/temp

12$ x=‘prog‘
```

▼ **Homework 5.10: shell's-eye view**

Which of the following words will the shell think are names of programs, arguments, or names of variables?

```
1$ blah1 blah2 blah3
2$ blah1 blah2 blah3 < blah4 > blah5
3$ blah1 blah2 blah3 | blah4 blah5 blah6
4$ blah1 blah2 blah3 && blah4 blah5 blah6
5$ blah1 blah2 blah3 || blah4 blah5 blah6

6$ blah1 blah2 ‘blah3‘
7$ blah1 ‘blah2 blah3‘
8$ blah1 ‘blah2 | blah3‘
9$ blah1 ‘blah2 && blah3‘
10$ blah1 ‘blah2 || blah3‘

11$ blah1 $blah2
12$ blah1=blah2
13$ blah1 = blah2
14$ blah1=‘blah2‘
15$ blah1 = ‘blah2‘
```

▲

**Why a Unix program should specify no output file: pp. 130−131**

```
#!/bin/ksh
#classmates: output login name of everyone in class who’s logged in.

~mm64/bin/roster 45 | sort > ~/inclass
who | awk ’{print $1}’ | sort | uniq > ~/loggedin
comm -12 ~/inclass ~/loggedin
rm ~/inclass ~/loggedin
exit 0
```

```
#!/bin/ksh
#classmates2: output to the file ~/classmates.out the login name of
#everyone in class who’s logged in.

~mm64/bin/roster 45 | sort > ~/inclass
who | awk ’{print $1}’ | sort | uniq > ~/loggedin
comm -12 ~/inclass ~/loggedin > ~/classmates.out
rm ~/inclass ~/loggedin
exit 0
```

Display the loginnames of the classmates who are logged in:

```
1$ classmates
```

```
2$ classmates2
3$ cat ~/classmates.out
4$ rm ~/classmates.out
```

Print the loginnames of the classmates who are logged in:

```
5$ classmates | lpr
```

```
6$ classmates2
7$ lpr ~/classmates.out
8$ rm ~/classmates.out
```

Write the loginnames of the classmates who are logged in into the file **~/people**:

```
9$ classmates > ~/people
```

```
10$ classmates2
11$ mv ~/classmates.out ~/people
```

Mail a letter to the classmates who are logged in:

```
12$ mail ‘classmates‘ < letter
```

```
13$ classmates2
14$ mail ‘cat ~/classmates.out‘ < letter
15$ rm ~/classmates.out
```

Do not write **| more** or **| lpr** at the end of a shellscript either, because the shellscript will not always send its output directly to the screen or printer.

```
#!/bin/ksh
#If the standard input of this shellscript comes from a keyboard
#and the standard output of this shellscript goes to a screen,
#then filter the standard output through more.

prog |
if [[ -t 1 ]]
then
    more
else
    cat
fi
```

An exception to this rule is a shellscript that directs it output to two or more different destinations. in this case, the names of all but one of them must be specified.

**Why a Unix program should specify no input file: pp. 130−131**

```
#!/bin/ksh
#not_responding: output the hosts in the standard input
#that did not respond to a ping.

sort > ~/locals

$S45/myping | sort |
comm -23 ~/locals - |
tr '.' ' ' |
sort +0n +1n +2n +3n |
tr ' ' '.'

rm ~/locals
exit 0
```

```
#!/bin/ksh
#not_responding2: output the hosts in the file ~/not_responding.in
#that did not respond to a ping.

sort ~/not_responding.in > ~/locals

$S45/myping | sort |
comm -23 ~/locals - |
tr '.' ' ' |
sort +0n +1n +2n +3n |
tr ' ' '.'

rm ~/locals
exit 0
```

Output the hosts in the file **hosts** that did not respond to a ping:

```
1$ not_responding < hosts
```

```
2$ cp hosts ~/not_responding.in
3$ not_responding2
4$ rm ~/not_responding.in
```

Output the hosts in the output of **prog** that did not respond to a ping:

```
5$ prog | not_responding
```

```
6$ prog > ~/not_responding.in
7$ not_responding2
8$ rm ~/not_responding.in
```

**Why a Unix program should output no header line: pp. 130−131**

The intended audience of any program will usually be another program. See Edward R. Tufte, *The Visual Display of Quantitative Information*, **http://www.edwardtufte.com/**, Chapter 6: Data Ink Maximization.

```
#!/bin/ksh
#classmates3: output login name of everyone in class who's logged in.

echo The following people are logged in:

~mm64/bin/roster 45 | sort > ~/inclass
who | awk '{print $1}' | sort | uniq > ~/loggedin
comm -12 ~/inclass ~/loggedin
rm ~/inclass ~/loggedin
exit 0
```

Count how many classmates are logged in:

```
1$ classmates | wc -l
```

```
2$ classmates3 | tail +2 | wc -l
```

Mail a letter to the classmates who are logged in:

```
3$ mail `classmates` < letter
```

```
4$ mail `classmates3 | tail +2` < letter
```

```
#!/bin/ksh
#Output login name of everyone in class who's logged in.

if [[ -t 1 ]]
then
    echo The following people are logged in:
fi

~mm64/bin/roster 45 | sort > ~/inclass
who | awk '{print $1}' | sort | uniq > ~/loggedin
comm -12 ~/inclass ~/loggedin
rm ~/inclass ~/loggedin
exit 0
```

**A World Wide Web gateway**

Most of the links in a World Wide Web page lead to other pages. But a link can also lead to a program. When you click on this link, the program will run and display its output in your browser. A program run from the Web is called a *gateway.* Put your gateways in your **~/public_html/cgi-bin** directory. The following gateway is named **classmates**.

To work correctly, a gateway must obey a set of rules called the *Common Gateway Interface*, or CGI. See **http://www.w3.org/CGI/**. For example, the first line of a gateway's standard output must begin with **Content-type:**. The **C** must be uppercase; there must be a dash, not an underscore; there must be no space on either side of the dash; and there must be a colon, not a semicolon. The second line of standard output must be empty. If the gateway runs a program without redirecting the program's standard output and standard error output (for example, **echo** and **comm** in the following gateway), then these outputs will be displayed in the web browser.

Since your gateway may be run by people other than you, it cannot use any of the variables created in your **.profile** file, e.g., your **$PATH**. In your gateway you must therefore write the full pathname of any command which is in an out-of-the-way directory, e.g. your personal **bin** or **cgi-bin** subdirectories. Or in your gateway you can say

```
echo '<P>'
echo My original '$PATH' was $PATH
export PATH=/home1/a/abc1234/bin:/home1/a/abc1234/public_html/cgi-bin:$PATH
echo '<BR>'
echo My new '$PATH' is $PATH
```

Until now, you have put all your executable files in your **~/bin** directory. But your gateway must go in your **~/public_html/cgi-bin** directory.

```
#!/bin/ksh
#Gateway script to output the loginnames of the classmates logged in.

echo Content-type: text/html
echo
echo '<HTML>'
echo '<HEAD>'
echo '<TITLE>Classmates logged in</TITLE>'
echo '</HEAD>'
echo '<BODY>'
echo '<H1>Classmates logged in</H1>'
echo '<PRE>'

~mm64/bin/roster 45 | sort > /tmp/inclass$$
who | awk '{print $1}' | sort | uniq > /tmp/loggedin$$
comm -12 /tmp/inclass$$ /tmp/loggedin$$
rm /tmp/inclass$$ /tmp/loggedin$$

echo '</PRE>'
echo '</BODY>'
echo '</HTML>'
exit 0
```

**chmod** your gateway **classmates** to **rwxr-xr-x** so that everyone in the world can execute it. In Homeworks 1.2 and 1.3 you already **chmod**'ed your home, **public_html**, and **cgi-bin** directories to **rwxr-xr-x**.

When testing your gateway, be sure to execute **~/public_html/cgi-bin/classmates**, not **~/bin/classmates**:

```
1$ cd ~/public_html/cgi-bin
2$ pwd

3$ classmates | more                                    the wrong shellscript
4$ ~/public_html/cgi-bin/classmates | more
5$ ./classmates | more          Dot stands for your current directory, Handout 1, p. 9.
```

The gateway should produce the following output. For **<PRE>** and **</PRE>**, see Handout 3, p. 10, lines 82 and 87.

```
Content-type: text/html

<HTML>
<HEAD>
<TITLE>Classmates logged in</TITLE>
</HEAD>
<BODY>
<H1>Classmates logged in</H1>
<PRE>
abc1234
def5678
ghi0912
</PRE>
</BODY>
</HTML>
```

Write the following in your **~/public_html/index.html** file:

```
<P>
Click
<A HREF = "http://i5.nyu.edu/cgi-bin/cgiwrap/abc1234/classmates">here</A>
to see the classmates logged in right now.
```

Because of the line

in the file **/etc/apache2/2.2/httpd.conf**, the link actually runs the gateway **/export/home/whitney/install-cgibin/cgiwrap-4.1/cgiwrap**

```
6$ cd /export/home/whitney/install-cgibin/cgiwrap-4.1
7$ ls -l cgiwrap
-rwxr-xr-x   1 root     root       67000 Oct 27  2011 cgiwrap
```

**cgiwrap** receives an environment variable named **$PATH_INFO** containing the string **abc1234/classmates**, which makes **cgiwrap** run your gateway **/home1/a/abc1234/public_html/cgi-bin/classmates**.

Better yet, all you need is a relative URL as in Handout 3, p. 30, ¶ (5):

```
<P>
Click
<A HREF = "/cgi-bin/cgiwrap/abc1234/classmates">here</A>
to see the classmates logged in right now.
```

```
Click here to see the classmates logged in right now.
```

When you click here, the browser will render the output of the gateway as

```
Classmates logged in

abc1234
def5678
ghi9012
```

**▼ Homework 5.11: write a gateway**

Write a gateway that doesn't always produce exactly the same output each time you run it. It can't be the gateway shown above that outputs the loginnames of the people in the class who are logged in now. It can't be the a stripped down version of this gateway, such as one that would output the loginnames of all the people who are logged in now. It could output

(1) the current date and time (let's not all do this one);

(2) a calendar of the current month;

(3) a picture of the current phase of the moon (Handout 2, p. 14, line 88);

(4) the number of people logged into i5.nyu.edu;

(5) the gateway's PID number;

(6) the names and contents of all the gateway's environment variables. Run the **env** program in Handout 1, p. 3, line 60; Handout 3, p. 16.

(7) the contents of just one environment variable, e.g. the gateway's **$PATH** or **$REMOTE_ADDR** variable. The latter is the IP address of the host that launched the gateway. ''Click here to see your IP address.'' Also output the IP address's hostname (use **nslookup**).

(8) whether or not you're logged into i5.nyu.edu (see ''Where is Mark logged into i5.nyu.edu right now?'' in his home page, and note that only a live human being, not a gateway, can give the lowercase **`-m`** option to **`who`**. A gateway would have to get the same effect by filtering the standard output of **`who`** through **`grep`**.);

(9) etc.

The gateway should not attempt to perform input—we'll do that later. Do only output. If you're outputting a picture (e.g., rows and columns of numbers and/or words, or a picture of the moon), don't forget to enclose it in the **`<PRE>`** and **`</PRE>`** in Handout 3, p. 10, lines 82 and 87.

Link your **`index.html`** to the gateway. Hand in a printout of your **`index.html`** file as rendered by your browser; the **`index.html`** file as displayed by your browser's **`View Source`**; your gateway (i.e., the shellscript); and the standard output of your gateway. Please circle the link in your home page that leads to your gateway.

▲

□