Summer 2013 Handout 4

Use tr (Handout 2, p. 17) as an adaptor between programs

```
1$ awk '69745 <= NR && NR <= 69746' $S45/Shakespeare.complete
QUINCE  Bless thee, Bottom! bless thee! thou art
    translated.

2$ awk '69745 <= NR && NR <= 69746' $S45/Shakespeare.complete |
tr '[a-z]' '[A-Z]'
QUINCE  BLESS THEE, BOTTOM! BLESS THEE! THOU ART
    TRANSLATED.

3$ prog1 | prog2
4$ prog1 | tr '[a-z]' '[A-Z]' | prog2
5$ prog1 | tr '[a-z] [A-Z]' | prog2
```
Can't give it one argument.

```
6$ compiler | assembler
7$ compiler | tr '$' '#' | assembler
```
We hoped we could do this, but we had to do this. Why do we need the quotes?

tr example: DNA

A strand of DNA is a long string of the letters **a**, **c**, **g**, and **t**:

```
1$ more $S45/dna
act
cat
tact
```

A strand of DNA can create another strand of the same length. The new strand has
a wherever the original had **t**;
t wherever the original had **a**;
c wherever the original had **g**;
g wherever the original had **c**.
Thus a strand such as **aagct** can create the strand **ttcga**. And then **ttcga** could create another copy of the original **aagct**.

The following command inputs a file of strands, one per line, and outputs the strands that they create.

```
2$ tr '[acgt]' '[tgca]' < $S45/dna | more
tga
gta
atga
```

To see the input and the output side by side on the screen, put the output into a file and display the input and output files simultaneously with the **pr** command. The **-i** option will make **pr** space with blanks, not tabs. (Some browsers do not render tabs correctly.) This option consists of the six characters minus, lowercase **i**, single quote, blank, single quote, one. The second option is minus lowercase L one. The **-w80** is minus lowercase W eighty.

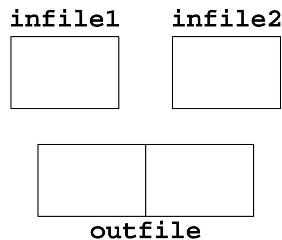
 All rights reserved ©2013 Mark Meretzky

```
3$ tr '[acgt]' '[tgca]' < $S45/dna > output
4$ pr -i' '1 -l1 -m -t -w80 $S45/dna output
act                     tga
cat                     gta
tact                    atga          etc.
```

If you filter data through the **tr '[acgt]' '[tgca]'** command twice, you will output a copy of the original data. Verify this by feeding the original and the twice-processed data to **cmp**:

```
5$ tr '[acgt]' '[tgca]' < $S45/dna | tr '[acgt]' '[tgca]' > output
6$ cmp $S45/dna output          compare: dead silence if identical (Handout 2, p. 12)
7$ rm output
```

Horizontal concatenation

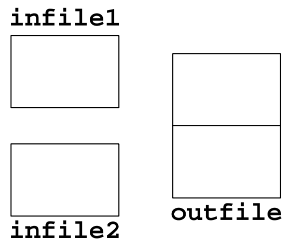


The first option consists of the six characters minus, lowercase **i**, single quote, blank, single quote, one. The second option is minus lowercase L one.

```
1$ pr -i' '1 -l1 -m -t -w80 infile1 infile2 > outfile
2$ ls -l outfile
```

Vertical concatenation

We did this in Handout 2, p. 24, line 13.




```
1$ cat infile1 infile2 > outfile
2$ ls -l outfile
```

▼ Homework 4.1: decryption

I typed a story into a file named **story** and then said

```
1$ tr '[ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz]' \
      '[BCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyzA]' < story \
      > $S45/encrypted
```

What I actually said was an easier way to write the above **tr**, with no space adjacent to the dashes and square brackets.

```
2$ tr '[A-Z][a-z]' '[B-Z][a-z]A' < story > $S45/encrypted
3$ rm story
```

Recreate the **story** file. Hand in the **tr** command you used to decrypt it. Side by side on the same page,

 All rights reserved ©2013 Mark Meretzky

use **pr** to print the first 38 lines of **$S45/encrypted** on the left and the first 38 lines of your decrypted reconstruction on the right. Give **pr** the five options **-i' '1 -l1 -m -t -w80** that you saw above. Use **head** only once; give it the **-38** argument. Don't use **deroff** or **tail**. The output will look better in a monospace font.

Assignment statements in the shell language

When giving a new value to a shell variable, do not write a **$** in front of its name. Write the **$** only when you *use* the value of a shell variable. See pp. 36–38, 88–89; **ksh**(1) pp. 8–9.

```
#!/bin/ksh
#Put strings into variables.

x=hello                 #No space around the equal sign: p. 36.
echo $x                 #Verify that you put "hello" into the variable.
echo x                  #What happens if you omit the dollar sign?

cf='chow fon'           #If the string contains blanks or tabs, quote it.
echo $cf
```

Why there must be no space on either side of the equal sign

	right	*wrong*
our point of view	**x=hello**	**x = hello**
computer's point of view	**blah=blah**	**blah = blah**

With white space on either side of the equal sign in **x = hello**, the computer would think that you were trying to run a program named **x** with two command line arguments, **=** and **hello**. See p. 88.

Why there must be a dollar sign before a variable: p. 88

In most programming languages, the quotation marks tell you which words are the names of variables and which words are strings:

```
x = "hello"          /* "hello" is a string of characters. */
x = hello            /* hello is the name of a variable. */
```

But in a language in which quotes are optional and variables are not declared, there would be no way to tell which words are the names of variables.

Use only a part of a variable's value

See **ksh**(1) pp. 10–12; and the tail modifier **:t** in **csh**(1) pp. 4, 6. Chop off the . . .

	prefix	*suffix*
shortest	#	%
longest	##	%%

The ***** is a wildcard. ***/** means "everything up to and including a slash";
.* means "a dot and everything after it".

 All rights reserved ©2013 Mark Meretzky

```
#!/bin/ksh

directory=/home1/a/abc1234
echo $directory          #echoes /home1/a/abc1234
echo ${directory#*/}     #echoes home1/a/abc1234
echo ${directory##*/}    #echoes abc1234
echo $directory          #still echoes /home1/a/abc1234

host=i5.nyu.edu
echo $host               #echoes i5.nyu.edu
echo ${host%.*}          #echoes i5.nyu
echo ${host%%.*}         #echoes i5
echo $host               #still echoes i5.nyu.edu
```

See the **##*/** in Handout 2, p. 13, lines 39–42; Handout 4, p. 21, line 36; **$PWD** in **ksh**(1) pp. 13, 17.

```
1$ cd
2$ pwd
/home1/a/abc1234

3$ echo $PWD
/home1/a/abc1234

4$ echo ${PWD#*/}
home1/a/abc1234

5$ echo ${PWD##*/}
abc1234
```

Two kinds of variables: pp. 38, 91–92

For environment variables, see Handout 3, p. 15. **$PATH** is an environment variable and **$PS1** (PS one) is a local variable; they were created in your **.profile** file in Handout 2, p. 13, lines 7–8 and 39–42. The Unix convention is to give uppercase names to environment variables, lowercase to local variables. In the Bourne shell language, creating an environment variable requires two statements:

	Korn shell and Bourne-Again shell	*Bourn shell*	*C Shell*
local	**x=hello**	**x=hello**	**set x = hello**
environment	**export X=hello**	**X=hello** **export X**	**setenv X hello**

To remove either kind of variable in any of the above shells, say **unset x**. Environment variables can be used in programs in any language, not just the shell language:

http://i5.nyu.edu/~mm64/INFO1-CE9545/src/getenv

```
#!/bin/ksh

echo $HOME
echo $PRINTER   #created in Handout 2, p. 13, lines 26-27
```

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

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

 All rights reserved ©2013 Mark Meretzky

```
 2 #include <stdlib.h>   /* for getenv, EXIT_SUCCESS, EXIT_FAILURE */
 3
 4 int main()
 5 {
 6     /* Print the value of $HOME, p. 199. */
 7
 8     const char *const p = getenv("HOME");
 9     if (p == NULL) {
10         return EXIT_FAILURE;
11     }
12
13     printf("%s\n", p);
14     return EXIT_SUCCESS;
15 }
```

```
1$ man -s 3c getenv
2$ gcc -o ~/bin/getenv getenv.c
```

C functions (`printf`, `getenv`) in section `3c`.
minus lowercase O to create `~/bin/getenv`

```
3$ ls -l ~/bin/getenv
-rwx------   1 abc1234     users       7080 May 28 15:58 /home1/a/abc1234/bin/getenv
```

```
4$ getenv
/home1/m/mm64
```

run `~/bin/getenv`

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

```
 1 #include <iostream>   //C++
 2 #include <cstdlib>    //for getenv, EXIT_SUCCESS, EXIT_FAILURE
 3 using namespace std;
 4
 5 int main()
 6 {
 7     if (const char *const p = getenv("HOME")) {
 8         cout << p << "\n";
 9         return EXIT_SUCCESS;
10     }
11
12     return EXIT_FAILURE;
13 }
```

```
5$ man -s 3c++ cout
6$ g++ -o ~/bin/getenv getenv.C
```

C++ objects (`cin`, `cout`) in section `3c++`.
minus lowercase O to create `~/bin/getenv`

```
7$ ls -l ~/bin/getenv
-rwx------   1 mm64     users       8444 May 28 15:58 /home1/a/abc1234/bin/getenv
```

```
8$ getenv
/home1/m/mm64
```

run `~/bin/getenv`

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

```
 1 public class Getenv {
 2     public static void main (String[] args) {
 3         final String home = System.getenv("HOME");
```

 All rights reserved ©2013 Mark Meretzky

```
 4         if (home == null) {
 5             System.exit(1);
 6         }
 7
 8         System.out.println(home);
 9         System.exit(0);
10     }
11 }
```

```
9$ cd ~/bin
10$ javac -help                      Java compiler
11$ javac -d . $S45/Getenv.java      Create Getenv.class.

12$ ls -l Getenv.class
-rw-------   1 abc1234     users        579 May 28 15:58 Getenv.class

13$ java Getenv                      Feed Getenv.class to the Java interpreter java.
/home1/m/mm64

14$ echo $?
0
```

You must turn on the **r** and **x** bits of a script in Perl, Python, PHP, or Ruby, just like a shellscript.

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

```
#!/bin/perl
#This script is written in Perl.  %ENV is an associative array.
#awk has associative arrays, too.  See textbook, pp. 123-124.

$home = $ENV{HOME};
if (!defined $home) {
    exit 1;
}

print "$home\n";
exit 0;
```

```
15$ man perl
16$ man -M /usr/perl5/man perlfunc          See p. 51 for print.
17$ man -M /usr/perl5/man perlvar           See p. 15 for %ENV.
18$ man -M /usr/perl5/man perlop            See p. 11 for \n.
19$ getenv.pl
/home1/m/mm64
```

 All rights reserved ©2013 Mark Meretzky

http://i5.nyu.edu/~mm64/INFO1-CE9545/src/getenv.py

```
#!/bin/python
import os    #the operating system module
import sys   #the system module

if os.environ.has_key('HOME'):
  print os.environ['HOME']
  sys.exit() #successful termination

sys.exit("environment variable HOME does not exist")
```

```
20$ getenv.py
/home1/m/mm64
```

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

```
#!/usr/php/5.2/bin/php -q
<?php

$home = getenv('HOME');
if (!$home) {
        exit(1);
}

echo $home . "\n";
exit(0);
?>
```

```
21$ getenv.php
/home1/m/mm64
```

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

```
#!/bin/ruby
#This script is written in Ruby.
#ENV is an object that takes the same [ ] as a hash.

home = ENV['HOME']
if home.nil?
  exit 1
end

puts home
exit 0
```

```
22$ /home1/m/mm64/public_html/INFO1-CE9970/local/bin/ri IO.puts
23$ getenv.rb
/home1/m/mm64
```

doc

 All rights reserved ©2013 Mark Meretzky

Loops in a shellscript

When you give a new value to a shell variable, don't write a **$** in front of its name. Write the **$** only when you *use* the value of a shell variable. See pp. 94–97, 144; **ksh**(1) p. 2; and Donald E. Knuth's article "The Complexity of Songs" in *SIGACT News* Volume 9 Number 2 (Summer 1977), pp. 17–24, at

http://portal.acm.org/citation.cfm?doid=358027.358042

```
#!/bin/ksh

echo Where have all the flowers gone?
echo Where have all the young girls gone?
echo Where have all the husbands gone?
echo Where have all the soldiers gone?
```

```
Where have all the flowers gone?
Where have all the young girls gone?
Where have all the husbands gone?
Where have all the soldiers gone?
```

```
#!/bin/ksh

for things in flowers 'young girls' husbands soldiers
do
    echo Where have all the $things gone?
done
```

If the list of words after the **in** doesn't fit on one line, you can continue it onto another line with a backslash. Make sure there is nothing after the backslash, not even a blank.

```
#!/bin/ksh
#Who copied the new .profile file into their home directory?

for loginname in yb610 ic297 bc1478 sum208 mm64 jp3195 up244 \
    aw1312 rz665
do
    ls -l ~$loginname/.profile   #~$loginname is their home directory
done
```

```
-r--r--r--   1 yb610    users       2975 Jun 10  2012 /home1/y/yb610/.profile
-r--r--r--   1 ic297    users       2975 May 29  2012 /home1/i/ic297/.profile
-r--r--r--   1 bc1478   users       2975 May 29  2012 /home1/b/bc1478/.profile
-r--r--r--   1 sum208   users       2975 May 29  2012 /home1/s/sum208/.profile
-rw-r--r--   1 mm64     users       2224 Sep 19  2012 /home1/m/mm64/.profile
```

Never write a loop that always iterates exactly once.

A *word* is an island surrounded by whitespace. The filename **/usr/dict/words** to the right of the following **in** is therefore one word and the loop iterates exactly once.

 All rights reserved ©2013 Mark Meretzky

```
#!/bin/ksh

for filename in /usr/dict/words
do
    grep separate $filename
done
```

```
#!/bin/ksh
#A simpler way to do the same thing.

grep separate /usr/dict/words
```

Nested loops: the crown jewels

The number of **do**'s must be equal to the number of **done**'s. Can you output the Republicans before the Democrats? Can you output the liberals of all parties first, followed by the moderates, followed by the conservatives?

http://i5.nyu.edu/~mm64/INFO1-CE9545/src/cartesian

```
#!/bin/ksh
#Output all 6 combinations, i.e., the Cartesian product.

for party in Democratic Republican
do
    for leaning in liberal moderate conservative
    do
        echo $party $leaning
    done
done
```

```
Democratic liberal
Democratic moderate
Democratic conservative
Republican liberal
Republican moderate
Republican conservative
```

The name of a variable can contain letters (upper and lowercase), digits, and underscores. See "identifiers" in **ksh**(1) p. 1. You therefore need {curly braces} around the name of any variable that is immediately followed by a letter, digit, or underscore. See p. 148–149; **ksh**(1) pp. 8–9.

```
#!/bin/ksh

for party in Democratic Republican
do
    for leaning in liberal moderate conservative
    do
        echo $party ${leaning}s
    done
done
```

 All rights reserved ©2013 Mark Meretzky

```
Democratic liberals
Democratic moderates
Democratic conservatives
Republican liberals
Republican moderates
Republican conservatives
```

Do “We Shall Overcome”, “Gimme that Old Time Religion” from *Inherit the Wind,* etc. like this:

```
#!/bin/ksh
#Output the three choruses of "If I Had a Hammer".

for verb in hammer ring sing
do
    for noun in danger warning ’love between my brothers and my sisters’
    do
        echo Id $verb out $noun       #no apostrophe
    done

    echo Aaaaaaaall over this laaaaaaand.
    echo          #Output an empty line after each chorus, p. 79.
    sleep 5       #Do nothing for five seconds, p. 73.
done
```

```
Id hammer out danger
Id hammer out warning
Id hammer out love between my brothers and my sisters
Aaaaaaaall over this laaaaaaand.

Id ring out danger
Id ring out warning
Id ring out love between my brothers and my sisters
Aaaaaaaall over this laaaaaaand.

Id sing out danger
Id sing out warning
Id sing out love between my brothers and my sisters
Aaaaaaaall over this laaaaaaand.
```

▼ **Homework 4.2: output every combination of one from group A and one from group B**

Do all of the following, but hand in only one. You get no credit if you hand in more than one.

(1) Write a shellscript with nested loops whose output is

```
========Beef======================
Beef with Black Bean Sauce
Beef with Broccoli
Beef with Garlic Sauce
Beef with Mixed Chinese Vegetables
Beef with Snow Peas
```

 All rights reserved ©2013 Mark Meretzky

```
========Chicken=====================
Chicken with Black Bean Sauce
Chicken with Broccoli
Chicken with Garlic Sauce
Chicken with Mixed Chinese Vegetables
Chicken with Snow Peas

========Pork=====================
Pork with Black Bean Sauce
Pork with Broccoli
Pork with Garlic Sauce
Pork with Mixed Chinese Vegetables
Pork with Snow Peas

========Shrimp=====================
Shrimp with Black Bean Sauce
Shrimp with Broccoli
Shrimp with Garlic Sauce
Shrimp with Mixed Chinese Vegetables
Shrimp with Snow Peas
```

You get no credit if the word **with** is written more than once in this shellscript.

(2) Write a shellscript named **fonts** with nested loops, whose output is

```
<P>
<BR><FONT SIZE = 1><EM>Size 1 in EM tags</EM></FONT>
<BR><FONT SIZE = 1><STRONG>Size 1 in STRONG tags</STRONG></FONT>
<BR><FONT SIZE = 1><CODE>Size 1 in CODE tags</CODE></FONT>
</P>

<P>
<BR><FONT SIZE = 2><EM>Size 2 in EM tags</EM></FONT>
<BR><FONT SIZE = 2><STRONG>Size 2 in STRONG tags</STRONG></FONT>
<BR><FONT SIZE = 2><CODE>Size 2 in CODE tags</CODE></FONT>
</P>

<P>
<BR><FONT SIZE = 3><EM>Size 3 in EM tags</EM></FONT>
<BR><FONT SIZE = 3><STRONG>Size 3 in STRONG tags</STRONG></FONT>
<BR><FONT SIZE = 3><CODE>Size 3 in CODE tags</CODE></FONT>
</P>

<P>
<BR><FONT SIZE = 4><EM>Size 4 in EM tags</EM></FONT>
<BR><FONT SIZE = 4><STRONG>Size 4 in STRONG tags</STRONG></FONT>
<BR><FONT SIZE = 4><CODE>Size 4 in CODE tags</CODE></FONT>
</P>

<P>
<BR><FONT SIZE = 5><EM>Size 5 in EM tags</EM></FONT>
<BR><FONT SIZE = 5><STRONG>Size 5 in STRONG tags</STRONG></FONT>
<BR><FONT SIZE = 5><CODE>Size 5 in CODE tags</CODE></FONT>
</P>
```


All rights reserved ©2013 Mark Meretzky

```
<P>
<BR><FONT SIZE = 6><EM>Size 6 in EM tags</EM></FONT>
<BR><FONT SIZE = 6><STRONG>Size 6 in STRONG tags</STRONG></FONT>
<BR><FONT SIZE = 6><CODE>Size 6 in CODE tags</CODE></FONT>
</P>

<P>
<BR><FONT SIZE = 7><EM>Size 7 in EM tags</EM></FONT>
<BR><FONT SIZE = 7><STRONG>Size 7 in STRONG tags</STRONG></FONT>
<BR><FONT SIZE = 7><CODE>Size 7 in CODE tags</CODE></FONT>
</P>
```

Put **"**double quotes**"** around all the text that you echo, to prevent the **<**'s and **>**'s from telling the shell to perform file I/O:

```
1$ echo "<This is how you can output $variables and angle brackets>"
```

Use **>>** to append the output of the shellscript to your **index.html** file. Then use **vi** to move the **</BODY>** and **</HTML>** tags down to the bottom of the file.

(3) Write a shellscript with nested loops freely based on Hamlet II, ii, 387–390 (lines 138,703–138,709 in **$S45/Shakespeare.complete**):

```
2$ awk '138703 <= NR && NR <= 138709' $S45/Shakespeare.complete
LORD POLONIUS    The best actors in the world, either for tragedy,
    comedy, history, pastoral, pastoral-comical,
    historical-pastoral, tragical-historical, tragical-
    comical-historical-pastoral, scene individable, or
    poem unlimited: Seneca cannot be too heavy, nor
    Plautus too light. For the law of writ and the
    liberty, these are the only men.
```

(4) Write anything with nested loops.

Non-song examples

```
#!/bin/ksh
#Display the status of the i5.nyu.edu laser printers.

echo edlab:
lpq -Pedlab | tail -3

echo ndlab:
lpq -Pndlab | tail -3
```

```
#!/bin/ksh
#An easier way to do the same thing.

for printer in edlab ndlab
do
    echo $printer:
    lpq -P$printer | tail -3
done
```

The following command will let each recipient know the names of all the others:

 All rights reserved ©2013 Mark Meretzky

```
1$ mail abc1234 def5678@ibm.com ghi9012@apple.com jkl3456@un.org < ~/letter
```

To make each person believe that they are the sole recipient, you must say

```
2$ mail abc1234 < ~/letter
3$ mail def5678@ibm.com < ~/letter
4$ mail ghi9012@apple.com < ~/letter
5$ mail jkl3456@un.org < ~/letter
```

An easier way to do this is

```
#!/bin/ksh
#Make each person believe they're the sole recipient.

for address in abc1234 def5678@ibm.com ghi9012@apple.com jkl3456@un.org
do
    mail $address < ~/letter
done
```

We'll get the email addresses from a file when we do `back quotes`.

Loop through all the files in a directory

```
#!/bin/ksh
#Output the login name of each student in the class:
#the first line of each biography file.

cd ~mm64/public_html/INFO1-CE9545/bio

for filename in *
do
    head -1 $filename
done
```

```
#!/bin/ksh
#Output the login name of each student in the class:
#the first line of each biography file.

for filename in ~mm64/public_html/INFO1-CE9545/bio/*
do
    head -1 $filename
done
```

```
yb610
bc1478
sum208
jp3195
up244
```

 All rights reserved ©2013 Mark Meretzky

```
#!/bin/ksh
#Encrypt the digits on lines 5 and 6 of each bio file (the home and
#work phone numbers).  You have no permission to do this.

for filename in ~mm64/public_html/INFO1-CE9545/bio/*
do
    echo $filename           #so I can watch the progress on the screen

    head -4 $filename > ~/temp

    awk '5 <= NR && NR <= 6' $filename |
        tr '[0123456789]' '[8906517243]' >> ~/temp

    tail +7 $filename >> ~/temp
    mv ~/temp $filename      #replace original file with new one
done
```

The three commands that create the **~/temp** file in the above shellscript can be written as a single **sed** command; see the **y** in Table 4.2 on p. 113.

```
sed '5,6y/0123456789/8906517243/' $filename > ~/temp
```

Travel to many directories

My Handouts on the web are PDF files in several directories:

```
~mm64/public_html/INFO1-CE9545/handout
~mm64/public_html/INFO1-CE9547/handout
~mm64/public_html/book/
```
etc.

Each time around the loop, the variable **$dirname** will contain the full pathname of a different one of these directories:

```
#!/bin/ksh
#Make sure that everyone has permission to read the Handouts.

cd ~mm64
chmod 755 . public_html   #chmod these two directories to rwxr-xr-x

for dirname in ~mm64/public_html/INFO1-CE9*/handout ~mm64/public_html/book
do
    cd $dirname
    chmod 755 .     #chmod each Handout directory to rwxr-xr-x
    chmod 444 *.pdf #chmod the PDF files in the directory to r--r--r--
done
```

 All rights reserved ©2013 Mark Meretzky

Travel to many directories with nested loops

```
#!/bin/ksh
#For each student, see if their 4 directories were chmod'ed correctly.

for loginname in yb610 ic297 bc1478 sum208 mm64 jp3195 up244 \
	aw1312 rz665
do
	cd ~$loginname

	for dirname in . bin public_html public_html/cgi-bin
	do
		ls -ld $dirname
	done
done
```

```
drwxrwxrwx   4 yb610    users         35 Aug  5  2012 .
drwxr-xr-x   2 yb610    users         20 Aug  5  2012 bin
drwxr-xr-x   3 yb610    users          9 Jul 24  2012 public_html
drwxr-xr-x   2 yb610    users          3 May 29  2012 public_html/cgi-bin
drwx--x--x   8 ic297    users         29 Dec 29 17:22 .
drwxr-xr-x   2 ic297    users         13 Jul 26  2012 bin
drwxr-xr-x   3 ic297    users          6 Jun 26  2012 public_html
drwxr-xr-x   2 ic297    users          2 May 29  2012 public_html/cgi-bin
drwxr-xr-x   5 bc1478   users         20 Jul 10  2012 .
```

Redirect all the output produced by a loop: p. 95

The following output file is destroyed and re-created during each trip around the loop. You end up with an output file that contains only one line, giving the loginname of the last person:

```
#!/bin/ksh

for filename in ~mm64/public_html/INFO1-CE9545/bio/*
do
	head -1 $filename > ~/inclass
done
```

```
rz665
```

The following output file is not destroyed and re-created during each trip around the loop. It grows one line longer during each trip:

```
#!/bin/ksh
#Output the loginname of each student in the class
#into the file ~/inclass.

for filename in ~mm64/public_html/INFO1-CE9545/bio/*
do
	head -1 $filename
done > ~/inclass
```

 All rights reserved ©2013 Mark Meretzky

```
yb610
bc1478
sum208
jp3195
up244
```
etc.

```
#!/bin/ksh

for filename in ~mm64/public_html/INFO1-CE9545/bio/*
do
    head -1 $filename
done | sort | uniq > ~/inclass
```

```
aw1312
bc1478
jp3195
rz665
sum208
```

Output one or more lines from the interior of a file: p. 116

```
1$ awk '{print $3}'                     Print the third field of every line.
2$ awk 'NR == 2 {print $3}'             Print the 3rd field of only line 2; see p. 120 for ==.   'pattern {action}'

3$ awk 'NR == 2 {print $0}'             Print all of line 2 only.
4$ awk 'NR == 2 {print}'                Print all of line 2 only: p. 114.
5$ awk 'NR == 2'                        Print all of line 2 only: p. 114.
6$ awk 'NR == 139069' $S45/Shakespeare.complete

7$ date
Tue May 28 15:58:15 EDT 2013

8$ date | awk '{print $2}'              prints May
9$ cal | awk 'NR == 1 {print $1}'       prints May
```

Even if **nslookup** is not in any of the directories on our **$PATH**, we can run it anyway if we tell the computer where to find it:

```
10$ /usr/sbin/nslookup i5.nyu.edu | awk 'NR == 5 {print $NF}'
128.122.109.53

11$ awk -F: '$4 == 15' /etc/passwd | more        variable doesn't have to be NR
sh2895:x:23158:15:Shintaro Hashimoto:/home1/s/sh2895:/usr/local/etc/suspendedshell
```

See Handout 3, pp. 26–27, for the comma:

```
12$ awk -F: '$4 == 15 {print $1, $5}' /etc/passwd | more
sh2895 Shintaro Hashimoto

13$ awk -F: '$4 == 15 {print $1 $5}' /etc/passwd | more
sh2895Shintaro Hashimoto
```

 All rights reserved ©2013 Mark Meretzky

```
14$ cd /usr/bin
15$ ls -l | tail +2 | awk '$2 > 1' | more      files with multiple hard links
-r-xr-xr-x  39 root     bin        12124 Jul  5  2012 adb
-r-xr-xr-x  18 root     bin        25888 Jul  5  2012 alias
-r-xr-xr-x   4 root     bin        60116 Jul  5  2012 apropos
```

Patterns with "and", "or", and regular expressions

```
1$ awk '10 <= NR && NR <= 20'                              p. 120 for <= and &&
2$ awk '127702 <= NR && NR <= 127715' $S45/Shakespeare.complete

3$ ls -l | tail +2 | grep '^-' | awk '1000 <= $5 && $5 < 2000 {print $NF}'
4$ ls -l | awk 'NR >= 2 && /^-/ && 1000 <= $5 && $5 < 2000 {print $NF}'

5$ /usr/sbin/nslookup 128.122.109.53 | awk '/name = / {print $NF}'
I5.HOME.NYU.EDU.
```

Because of the **$1 ~**, the **$** in **/\.ssh$/** means the end of **$1**, not the end of the line.

```
6$ netstat -a -f inet -P tcp | awk '2 <= NR && NR <= 4 || $1 ~ /\.ssh$/' | head -6
TCP: IPv4
   Local Address        Remote Address    Swind Send-Q Rwind Recv-Q    State
-------------------- -------------------- ----- ------ ----- ------ -----------
i5.ssh               3A_IMAC_03.NDLAB.ITS.NYU.EDU.49630 131008      0 128872      0 ESTABLISHED
      *.ssh                *.*                0      0 128000      0 LISTEN
i5.ssh               3A_IMAC_03.NDLAB.ITS.NYU.EDU.50562 131024      0 128872      0 ESTABLISHED
```

When not to use awk

A special-purpose tool such as **head** is faster and simpler than a multi-purpose, programmable tool such as **awk**.

```
1$ awk 'NR == 1'              The fox knows many little things; the hedgehog knows one big thing.
2$ head -1                    faster and simpler way to do the same thing

3$ awk 'NR <= 10'
4$ head -10                   faster and simpler way to do the same thing
5$ head                       even simpler: -10 is the default (Handout 3, p. 26)

6$ awk 'NR >= 13'
7$ tail +13                   faster and simpler way to do the same thing
```

Patterns that compare two strings

Like many languages, awk requires **"**double quotes**"** around a string.

```
1$ awk -F: '$4 == 15 {print $1}' /etc/passwd | more
2$ awk -F: '$7 == "/bin/ksh" {print $1}' /etc/passwd | more
```

 All rights reserved ©2013 Mark Meretzky

```
3$ awk -F: '$7 != "/bin/ksh" {print $7}' /etc/passwd | sort | uniq -c
   6 /bin/bash
 121 /bin/csh
   1 /bin/tcsh
  19 /usr/bin/bash
   1 /usr/bin/csh
   1 /usr/bin/pfksh
   1 /usr/bin/pfsh
   1 /usr/lib/uucp/uucico
3931 /usr/local/etc/expiredshell
   3 /usr/local/etc/suspendedshell

4$ awk '$1 == "UserDir" {print $2}' /etc/apache2/2.2/httpd.conf

5$ cd /dev/pts                            On other systems, /dev or /devices/pseudo
6$ pwd
7$ ls -l | awk 'NR >= 2 && $3 != "root"' | head -5
crw--w----   1 mm64     tty       36,  1 May 28 15:56 1
crw--w----   1 yc801    tty       36,  2 Apr  8 18:19 2
crw--w----   1 esh322   tty       36,  3 Apr  7 23:51 3
crw--w----   1 yc801    tty       36,  5 Apr  8 18:51 5
crw--w----   1 yc801    tty       36,  6 Apr  8 18:19 6
```

Sort the output of **awk** into increasing numeric order, ignoring the first 9 fields on each line. See p. 106.

```
8$ ls -l | awk 'NR >= 2 && $3 != "root"' | sort +9n | head -5
crw--w----   1 mm64     tty       36,  1 May 28 15:56 1
crw--w----   1 yc801    tty       36,  2 Apr  8 18:19 2
crw--w----   1 esh322   tty       36,  3 Apr  7 23:51 3
crw--w----   1 yc801    tty       36,  5 Apr  8 18:51 5
crw--w----   1 yc801    tty       36,  6 Apr  8 18:19 6

9$ grep '::' /etc/passwd
10$ grep '^[^:]*::' /etc/passwd                 people with no password: p. 103
11$ awk -F: '$2 == ""' /etc/passwd              people with no password: pp. 116–117
```

```
#!/bin/ksh
#Output the real name of everyone in the class, one per line.
#Each real name is the second line of a bio file.

for filename in ~mm64/public_html/INFO1-CE9545/bio/*
do
    awk 'NR == 2' $filename
done
```

```
Yevgeniy Berezovskiy
Barry chennankara
sarika
Jules Panopoulos
Uri Poltiyelov
```

 All rights reserved ©2013 Mark Meretzky

```
#!/bin/ksh
#Output the number of people in the class who work in area code 212.
#Each work number is the sixth line of a bio file.

for filename in ~mm64/public_html/INFO1-CE9545/bio/*
do
    awk 'NR == 6' $filename
done | grep 212 | wc -l
```

if-then-else: p. 140

Format and space an **if** statement *exactly* as shown below. There must be space before and after the **[[**, and before the **]]**. The number of **if**'s must be equal to the number of **fi**'s, just as the number of **do**'s must be equal to the number of **done**'s. See **ksh**(1) p. 2.

http://i5.nyu.edu/~mm64/INFO1-CE9545/src/iftest

```
#!/bin/ksh
a=10   #No space around the = sign!  Try it again with a=20.
b=20

if [[ $a -eq $b ]]
then
    echo They are equal.
fi
```

If the **then** and **fi** are not at the start of their lines, we will need a semicolon in front of them (pp. 140, 95),

```
if [[ $a -eq $b ]]; then echo They are equal.; fi
```

else and elif

These **if**'s are consecutive and mutually exclusive.

```
#!/bin/ksh
a=10
b=20

if [[ $a -eq $b ]]
then
    echo They are equal.
fi

if [[ $a -ne $b ]]
then
    echo They are not equal.
fi
```

 All rights reserved ©2013 Mark Meretzky

```
#!/bin/ksh
#An easier way to do the same thing.

a=10
b=20

if [[ $a -eq $b ]]
then
    echo They are equal.
else
    echo They are not equal.
fi
```

```
#!/bin/ksh
a=10
b=20

if [[ $a -eq $b ]]
then
    echo They are equal.
elif [[ $a -lt $b ]]
then
    echo a is less than b.
else
    echo a is greater than b.
fi
```

A Unix error message always begins with the name of the program, a colon, and a blank, followed by the message itself. See **perror**(3c); examples are on pp. 17, 57, 61, 67, 84, 92, 178, 182, 191, 207, 224, 240, 339.

```
#!/bin/ksh
day=2

if [[ $day -eq 3 ]]
then
    echo three french hens
elif [[ $day -eq 2 ]]
then
    echo two turtle doves
elif [[ $day -eq 1 ]]
then
    echo a partridge in a pair tree
else
    echo $0: illegal day $day
    exit    #destroys the local variable day
fi
```

 All rights reserved ©2013 Mark Meretzky

C, C++, Java, JavaScript, awk, C shell	**else if**
C and C++ preprocessors	**#elif**
Bourne, Korn, and Bourne-Again shells, Ruby	**elif**
Perl	**elsif**
Tcl	**elseif**
troff, nroff	**.el .if**
m4	**.ifelse**

Logical expressions in if, elif, and while statements

Within the **[[**square brackets**]]**, you need whitespace around each operator and operand. Compare two numbers with the following six relational operators familiar from Fortran:

-eq *equals*	**-lt** *less than—do not use* **<**	**-le** *less than or equals*
-ne *not equals*	**-ge** *greater than or equals*	**-gt** *greater than*

Compare two strings with

== *equals (used to be one* **=***)*	**<** *less than, i.e., comes before in alphabetical order*
!= *not equals*	**>** *greater than*

For environment variables such as **$LOGNAME**, see Handout 3, p. 16; p. 136 in the textbook; and **environ**(5). A **$LOGNAME** in a shellscript is the loginname of the person running the shellscript, not the loginname of the person who wrote the shellscript.

```
if [[ $# -eq 3 ]]                Compare two numbers; $# is number of command line arguments.
if [[ $LOGNAME == abc1234 ]]     Compare two strings. $LOGNAME is called $USER on other systems.

if [[ 007 -eq 7 ]]               true: both are the number seven
if [[ 007 == 7 ]]                false: three-character string and one-character string
```

The shell language is inconsistent with the **awk** language:

```
$a -eq $b        numerical comparison inside the [[double brackets]] of a shell if statement
NR == 10         numerical comparison inside the argument of awk
```

Use **&&** and **||** (for “and” and “or”) to build compound expressions. For example,

```
if [[ $LOGNAME != abc1234 && $LOGNAME != def5678 ]]
then
    echo $0: Only abc1234 and def5678 have permission to run this program.
    exit
fi

if [[ $word1 <= $word2 ]]                            wrong: ksh has no <=
if [[ $word1 < $word2 || $word1 == $word2 ]]         right
```

The complete list of useful goodies that can go in the **[[**square brackets**]]** is in **ksh**(1) pp. 19–22. For example,

```
if [[ -e name ]]            If a file or directory (or anything else) with this name exists
if [[ -f filename ]]        If a file with this name exists
if [[ -d dirname ]]         If a directory with this name exists
if [[ ! -f filename ]]      If no file with this name exists
```

 All rights reserved ©2013 Mark Meretzky

```
if [[ -r name ]]
```
If the person running this shellscript has **r** *permission for this file or directory*

```
if [[ -w name ]]
```
If the person running this shellscript has **w** *permission for this file or directory*

```
if [[ -x name ]]
```
If the person running this shellscript has **x** *permission for this file or directory*

```
if [[ -r name && -w name ]]
```

A shellscript that takes a command line argument

Almost every program takes command line arguments:

```
1$ ls -l
2$ cal 5 2013
3$ finger abc1234
4$ echo $m little monkeys jumping on the bed--
5$ grep Dunsinane $S45/Shakespeare.complete
```

`$1`	*the shellscript's first command line argument, p. 82*
`$2`	*the shellscript's second command line argument*
`$3`	*the shellscript's third command line argument, etc.*
`$*`	*all the command line arguments, p. 83*
`$#`	*the number of command line arguments, p. 135*
`$0`	*the name of the shellscript, p. 85; Handout 4, p. 20*

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

```
 1 #!/bin/ksh
 2 #Copy a file to the ~/public_html directory so it can be seen
 3 #on the web.  Sample use: post filename.html
 4
 5 if [[ $# -ne 1 ]]
 6 then
 7     echo $0: requires exactly one argument
 8     exit
 9 fi
10
11 if [[ ! -f $1 ]]
12 then
13     echo $0: there is no file named $1
14     exit
15 fi
16
17 if [[ ! -r $1 ]]
18 then
19     echo $0: you must have r permission for the file $1
20     exit
21 fi
22
23 if [[ ! -d ~/public_html ]]
24 then
25     echo $0: there is no directory named ~/public_html
26     exit
27 fi
28
29 #$newfilename is the full pathname of the copy we will create in ~/public_html.
```

 All rights reserved ©2013 Mark Meretzky

```
30 newfilename=~/public_html/${1##*/}   #Handout 4, pp. 3-4 for ##
31
32 if [[ -e $newfilename ]]
33 then
34     echo $0: $newfilename already exists and I will not harm it
35     exit
36 fi
37
38 chmod u+w ~/public_html  #Give cp permission to copy the file into ~/public_html.
39 cp $1 $newfilename
40 chmod 444 $newfilename             #r--r--r--
41 chmod a+rx ~ ~/public_html         #Turn on all three r's and all three x's.
```

$1 is the entire first command line argument. The expression **${1##*/}** in line 30 is the last word of the argument. If you typed an argument that contained a pathname,

```
6$ post /some/directory/filename.html
```

then **$1** would contain **/some/directory/filename.html**, the value of **${1##*/}** would be **filename.html**, the variable **$newfilename** in line 30 would contain **/home1/a/abc1234/public_html/filename.html**, and lines 39–40 would execute the commands

```
cp /some/directory/filename.html /home1/a/abc1234/public_html/filename.html
chmod 444 /home1/a/abc1234/public_html/filename.html
```

You could then point your browser at

```
http://i5.nyu.edu/~abc1234/filename.html
```

▼ Homework 4.3: create the ~/public_html directory

Hand in one copy of **post** incorporating the following three improvements.

(1) If there is no **~/public_html** directory, we execute the above lines 25–26. Replace these two lines with lines that will create the **~/public_html** directory if there is nothing else named **~/public_html**.

The new lines will output an error message and exit if anything else named **~/public_html** already exists (a file, hardware device, symbolic link, etc). The error message should be "a non-directory **~/public_html** already exists and I will not harm it". After that, the new lines will give the person running the shellscript permission to create a new subdirectory in the ~ directory, if they do not already have this permission. Finally, the new lines will create the directory **~/public_html**. Do not output any message confirming that you have created **~/public_html**: the new lines must produce no output other than error messages. The new lines must not exit after creating **~/public_html**; let the shellscript continue onwards.

(2) Insert an **if** after line 39 to output an error message and exit if the file **$newfilename** does not now exist. Should the **if** come before or after line 40? Do not output any message confirming that **$newfilename** does exist.

▲

while loop

For the **while** loop, see pp. 144–145 in the textbook; **ksh**(1) p. 2. For the **let** command, see **ksh**(1) pp. 18–19, 44.

 All rights reserved ©2013 Mark Meretzky

```
#!/bin/ksh
#Output the lyrics to "100 Bottles of Beer on the Wall".

b=100                         #No space around the equal sign.

while [[ $b -ge 1 ]]          #pp. 144-145
do
    echo $b bottles of beer on the wall
    echo $b bottles of beer
    echo If one of those bottles should happen to fall
    let b=b-1   #or let --b; spaces would need quotes: let 'b = b - 1'
    echo $b bottles of beer on the wall

    echo                      #Skip a line.
    sleep 2
done
```

Four ways to feed input into any Unix program

sort, **grep**, **sed**, **awk**, **lpr**, **cat**, etc., can all take their input from any one of these four sources. The first three are the various forms of standard input, to which **$#**, **$1**, **$2**, etc., are blind (Handout 2, p. 22).

```
1$                 sort                  standard input from the keyboard
2$                 sort < infile         standard input from one input file
3$ previousprog | sort                   standard input from a pipe

4$                 sort infile1 infile2 infile3   not standard input
```

The shellscripts we have written, however, can take their input from only the first three of the above sources. To let them draw input from the fourth,

```
#!/bin/ksh
#Copy each file named as a command line argument to the ~/public_html
#directory.  Sample use: post filename.html [filename.html ... ]

if [[ $# -le 0 ]]
then
    echo $0: requires at least one filename as command line argument
    exit
fi

for filename in $*            #The "in $*" is optional: pp. 144-145.
do
    #$newfilename is full pathname of file to be created.
    newfilename=~/public_html/${filename##*/}
    cp $filename $newfilename
    chmod 444 $newfilename
done
```


All rights reserved ©2013 Mark Meretzky

Redirect the standard input with <<

```
#!/bin/ksh
#Output the line in the file ~/myphone.data containing the command
#line argument.  Sample use: myphone Galt

if [[ $# -ne 1 ]]
then
    echo $0: requires one command line argument giving a name
    exit
fi

grep -i $1 ~/myphone.data
```

This file is **~/myphone.data**. It is not a shellscript; do not turn on its **x** bits.

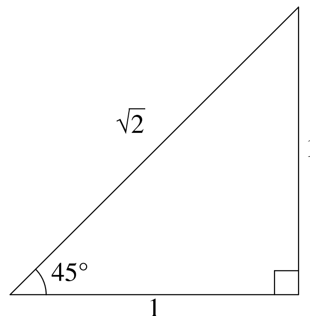
```
(212) 999-9999   John Galt
(215) 765-4321   Hank Rearden
(212) 211-1111   Howard Roark
```

Combine the above into a single file with **<<**. See pp. 93–94 in the textbook, **ksh**(1) p. 23.

```
#!/bin/ksh
#Output the line of data containing the command line argument.
#Sample use: myphone Galt

if [[ $# -ne 1 ]]
then
    echo $0: requires one command line argument giving a name
    exit
fi

grep -i $1 <<xyxyX7
(212) 999-9999   John Galt
(215) 765-4321   Hank Rearden
(212) 211-1111   Howard Roark
xyxyX7
```



Tangent is “opposite over adjacent”. To compute the value of π,

$$\tan 45° = \frac{\text{opposite}}{\text{adjacent}} = \frac{1}{1} = 1 \tag{1}$$

Another way to say the same thing is

 All rights reserved ©2013 Mark Meretzky

$$\arctan 1 = 45° \qquad (2)$$

Instead of measuring the size of the angle in degrees, we can use radians. One degree equals $\frac{\pi}{180}$ radians, so 45 degrees equals $\frac{\pi}{4}$ radians:

$$\arctan 1 = \frac{\pi}{4} \qquad (3)$$

Now you can see why we switched from degrees to radians: it introduced a π into the equation. To get the π all by itself, multiply both sides by 4:

$$4\arctan 1 = \pi \qquad (4)$$

You saw **bc** in Handout 2, p. 21. In the shell language, put a **#** in front of each comment. But in the standard input of **bc**, put **/*** and ***/** around each comment.

```
#!/bin/ksh
#Output the value of pi.  Comments in the lines of
#standard input fed to bc are surrounded with /* and */.

bc -l <<xyxyX7        #minus lowercase L for math library
scale = 66            /* number of digits to right of decimal point */
4 * a(1)              /* 4 times the arctangent of 1 */
xyxyX7
```

```
3.141592653589793238462643383279502884197169399375105820974944592304
```

Examine a program's exit status

Every Unix program returns an *exit status* number after it finishes. Zero indicates success; any other number (positive or negative) indicates failure. The exit status of the last command is held in the variable **$?** in the Korn shell language, the variable **$status** in the C Shell language. See p. 140; **ksh**(1) p. 14; Handout 2, p. 2.

```
1$ 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

2$ echo $?                            echo %errorlevel% in Windows Command Prompt
0                                     cal terminated successfully.

3$ cal 13 2013
cal: bad month                        cal outputs an error message
usage: cal [ [month] year ]
4$ echo $?
1                                     cal terminated unsuccessfully.
```

 All rights reserved ©2013 Mark Meretzky

```
5$ grep 'root' /etc/passwd            Look for the superuser's login name.
root:x:0:0:root@i5:/root:/usr/bin/bash
jh1997:x:16423:15:Jacqueline Harootian:/home1/j/jh1997:/usr/local/etc/expiredshell
6$ echo $?
0                                     grep terminated successfully.

7$ grep 'superuser' /etc/passwd       Look for nonexistent login name.
8$ echo $?
1                                     grep returns 1 to indicate it found nothing.

9$ grep 'root' /etc/password          Misspelled the filename.
grep: can't open /etc/password
10$ echo $?
2                                     grep returns 2 to indicate that file does not exist.
```

To return an exit status number, write the number after the **exit** keyword in a shellscript (pp. 141–142), or call the **exit** function in a C or C++ program (pp. 173–174).

▼ Homework 4.4: Examine the exit status of a few commands (not to be handed in)

Give a few commands and examine their exit status: **mv**, **mkdir**, **cmp -s** (Handout 2, p. 12), **mail -e** (Handout 2, p. 14, lines 79–83), **gcc** (Handout 3, p. 24), etc. Do they return 0 for success and non-zero for failure, e.g., trying to **mv** a non-existent file? Does the **man** page for each command mention its exit status number?

▲

Examine the exit status in an if statement: p. 140

An **if** statement is "true" when it consists of a command whose exit status is 0.

```
#!/bin/ksh
#Output the loginname of everyone who has a link in their home page.
#grep -i means "case insensitive"; see -y on p. 85 and grep(1).
#Produce exit status 0 if at least one person has a link, 0 otherwise.

status=1                #Guilty until proven innocent.

for loginname in yb610 ic297 bc1478 sum208 mm64 jp3195 up244 \
    aw1312 rz665
do
    grep -i HREF ~$loginname/public_html/index.html > /dev/null

    if [[ $? -eq 0 ]]
    then
        echo $loginname
        status=0
    fi
done

exit $status
```

```
bc1478
jp3195
up244
rz665
```

 All rights reserved ©2013 Mark Meretzky

```
#!/bin/ksh

status=1

for loginname in yb610 ic297 bc1478 sum208 mm64 jp3195 up244 \
    aw1312 rz665
do
    if grep -i HREF ~$loginname/public_html/index.html > /dev/null
    then
        echo $loginname
        status=0
    fi
done

exit $status
```

If the **then** and **fi** are not at the start of a line, we will need a semicolon in front of them (as in Handout 4, p. 17).

```
if grep -i HREF index.html > /dev/null; then echo $loginname; status=0; fi
```

The **-q** option of **/usr/xpg4/bin/grep** (p. 140) prevents it from producing any output: you get only the exit status. The following options all do the same thing:

```
1$ /usr/xpg4/bin/grep -q word filename      or -s on other versions of Unix
2$ cmp -s filename1 filename2               Handout 2, p. 12
3$ mail -e                                  Handout 2, p. 14, lines 79–83
```

If we change the **grep -i** in the above shellscript to **/usr/xpg4/bin/grep -iq**, we could remove the **> /dev/null**.

```
#!/bin/ksh

for loginname in yb610 ic297 bc1478 sum208 mm64 jp3195 up244 \
    aw1312 rz665
do
    grep -i HREF ~$loginname/public_html/index.html > /dev/null &&
        echo $loginname
done

exit 0      #No longer using exit status to indicate if found HREF.
```

For **&&** and **||**, see pp. 143–144; **ksh**(1) p. 1; Handout 2, p. 14, lines 79–83. Get me Mostly Mozart tickets *and* I'll love you forever. Get me Mostly Mozart tickets *or* I'll throw myself out the window. In C or C++,

```
if (a == b && c == d) {       /* will compare c and d only if a == b */
if (a == b || c == d) {       /* will compare c and d only if a != b */
```

▼ Homework 4.5: check the exit status in post

Remove the **if** that you added to the shellscript **post** after line 39 in Homework 4.3. Then insert three **if** statements in various places in **post** to check the exit status of the **mkdir**, **cp**, and **chmod**'s in **post**. Because of the exclamation point, the following **if** will be true if the **mkdir** produced a non-zero exit status.

 All rights reserved ©2013 Mark Meretzky

```
if ! mkdir ~/public_html
then
```

Output no message confirming that anything has worked correctly. Output nothing except error messages.
▲

Examine the exit status of the command after the keyword “while”

```
#!/bin/ksh
#Go into an infinite loop: "Paul Clifford" (1830)
#by Edward George Earle Bulwer-Lytton (1803-1873)

while true              #p. 147
do
     echo It was a dark and stormy night.
     echo Some Indians were sitting around a campfire.
     echo Then their chief rose and said,
     echo
     sleep 2
done
```

`while grep word file > /dev/null` *Keep looping as long as a certain word is in a file,*
`while who | grep abc1234 > /dev/null` *as long as* **abc1234** *is still loged in,*
`while ps -Af | tail +2 | grep progname > /dev/null` *as long as a certain program is still running.*

```
if false
then
     The commands on these lines
     will always be skipped.
fi
```

Four ways to put an image file into your public_html subdirectory

A **.gif** or **.jpg** file contains a digitized image. Remember to turn on all three of its **r** bits to see it on the World Wide Web.

(1) If you see a **.gif** file on i5.nyu.edu, simply copy it to your **~/public_html** directory. Then turn on all three of the file’s **r** bits if they are not already on.

```
1$ cd ~/public_html
2$ pwd

3$ cp $S45/construction.gif .

4$ ls -l construction.gif
-r--------   1 abc1234    users       541 May 28 15:58 construction.gif

5$ chmod 444 construction.gif          Give everyone permission to see it: r--r--r--

6$ ls -l construction.gif
-r--r--r--   1 abc1234    users       541 May 28 15:58 construction.gif
```

(2) If you know the URL of a **.gif** file on another host, you can copy it to your **~/public_html** directory like this:

 All rights reserved ©2013 Mark Meretzky

```
7$ cd ~/public_html
8$ pwd

9$ lynx -source http://www.nyu.edu/images/torch1.gif > torch1.gif
10$ ls -l torch1.gif
-rw-------   1 abc1234     users        835 May 28 15:58 torch1.gif

11$ chmod 444 torch1.gif            Give everyone permission to see it: r--r--r--

12$ ls -l torch1.gif
-r--r--r--   1 abc1234     users        835 May 28 15:58 torch1.gif
```

To verify that a GIF file arrived intact, don't peek beyond the first six bytes. The rest of a GIF file is binary. The **head** we get by default (**/bin/head**) does not have the **-c** option, so we use the other **head**. (You could also say **cut -c1-6**.)

```
13$ /usr/bin/ghead -c6 torch1.gif                    See the first six characters.
GIF87a

14$ man head                                  documentation for /bin/head
15$ man ghead                             documentation for /usr/bin/ghead
```

(3) Mail the picture to your **i5.nyu.edu** account as an attachment. Then use the **alpine** in Handout 2, p. 24 to save the enclosure in a file in your home directory.

```
16$ cd
17$ pwd

18$ alpine
L FOLDER LIST
I INBOX
down arrow  till you get to the letter with the attachment, then  RETURN
> View Attch
down arrow  till you get to the attachment, then  RETURN
S filename RETURN
Q
Really quit Alpine?
Y
19$ ls -lt | head
20$ chmod  the downloaded file to turn on all three of its r bits
21$ mv  the downloaded file to your ~/public_html directory
```

(4) Use the "secure file transfer program" **sftp**.

Four ways to put your image into a page

Let's assume that your image file is named **~/public_html/dog.gif**.

(1) To display the image in the page,

```
My dog in Colorado, 2013:
<IMG SRC = "dog.gif"
ALT = "[My dog in Colorado, 2013]">
Isn't he cute?
```

(2) To use the image as your page's background, change the **<BODY>** tag (Handout 3, p. 9, line 6) to

```
<BODY BACKGROUND = "dog.gif">
```

 All rights reserved ©2013 Mark Meretzky

(3) Instead of displaying the image in the page, you can make a link from your page to the image.

```
I took my
<A HREF = "dog.gif">dog</A>
to Colorado in 2013.
```

I took my dog to Colorado in 2013.

(4) Instead of asking the user to click on an underlined word, you can ask him or her to click on an image:

```
Click on the thumbnail picture of my dog for more information about him:
<BR>
<A HREF = "bigdog.html">
<IMG SRC = "littledog.gif"
ALT = "[My dog in Colorado, 2013]"></A>
```

□

 All rights reserved ©2013 Mark Meretzky