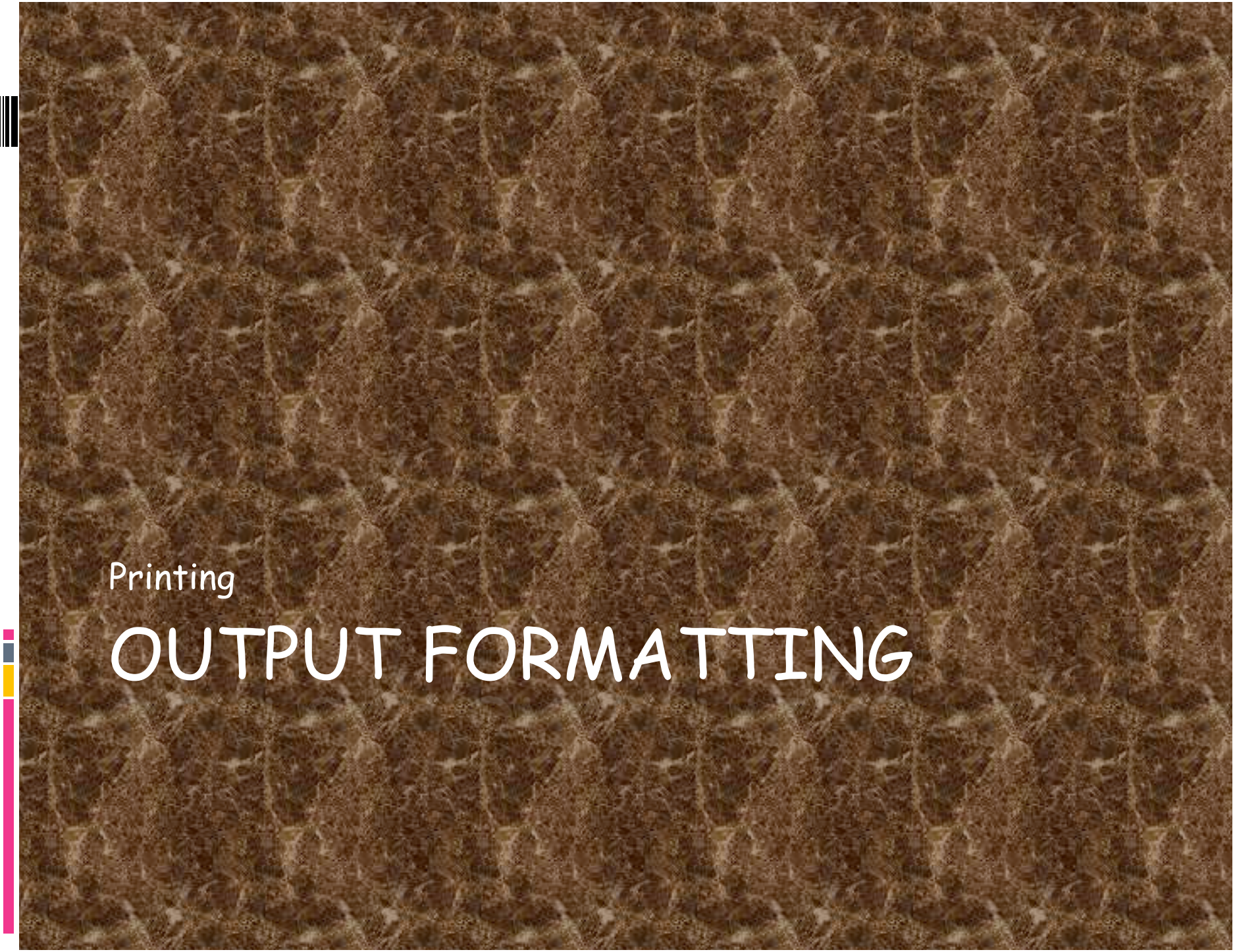
Printing
OUTPUT FORMATTING

printf: formatted printing

So far we have just been copying stuff from standard-in, files, pipes, etc to the screen or another file.

Say I have a file with names and phone numbers. I would like to print it out with vertically aligned columns.

(so my printout is not in the same format as the input file)

File contents:

```
Bob 4929
Chuck 4882
```

Desired display:

```
Bob     4929
Chuck   4882
```

printf

is a command from the C programming language to control printing.

Shell script

```
#!/bin/bash
printf "Hello world.\n"
a=`echo Hello world. | wc | nawk '{print $2}' `
printf "This phrase contains %d words\n" $a
```

Run it

```
alpaca.ceri.memphis.edu512:> printex.sh
Hello world.
This phrase contains 2 words
alpaca.ceri.memphis.edu513:>
```

```
#!/bin/bash
printf "Hello world.\n"
a=`echo Hello world. | wc | nawk '{print $2}' `
printf "This phrase contains %d words\n" $a
```

We need the double quotes " . . . " to define an argument (the stuff inside the quotes) for printf.

```
#!/bin/bash
printf "Hello world.\n"
a=`echo Hello world. | wc | nawk '{print $2}' `
printf "This phrase contains %d words\n" $a
```

The argument in double quotes has

- Regular text ("Hello world", "This text contains ")

- Some funny new thing - %d – a format specifier.

- The already known specification for a new line - \n

```
#!/bin/bash
printf "Hello world.\n"
a=`echo Hello world. | wc | nawk '{print $2}' `
printf "This phrase contains %d words\n" $a
```

We also have another argument, the $a, which is a shell variable, at the end.

Note that the items are delimited with spaces, not commas.

```
#!/bin/bash
printf "Hello world.\n"
a=`echo Hello world. | wc   nawk '{print $2}' `
printf "This phrase contains %d words\n" $a
```

We also have an example of nawk (which can be understood from the quick introduction given previously to allow nawk use in the homework).

Ignoring the details, this line assigns the value of the shell variable a to be the number of words in the string "Hello world." The variable a therefore contains an integer value (as a character string).

printf: format specifiers

How to specify the format for printing various types of things

```
printf "This phrase contains %d words\n" $a
```

We are going to print out what is in the double quotes.

printf: format specifiers

```
printf "This phrase contains %d words\n" $a
```

No problem for everything but the %d.

And what is that shell variable $a at the end?

printf: format specifiers

```
printf "This phrase contains %d words\n" $a
```

The shell variable a contains the number of words. We want this (number) information where the %d is located in the format specification.

The %d and the $a are "paired".

printf: format specifiers

```
printf "This phrase contains %d words\n" $a
```

The %d format specifier is used to control how contents of the shell variable, a, are printed.

printf: format specifiers

Specify how to print various types of things

%d signed decimal integer

(The word decimal means base 10, as opposed to octal - base 8, or hexadecimal - base 16, or a number with a decimal point. The word integer means a whole number, no decimal point and fraction).

printf: format specifiers

Modifying how decimal integers are printed.

`%<N>.<DIGITS>d`

says use a field N characters wide, with DIGITS digits (uses leading zeros, DIGITS can be > N (N gets overridden), or DIGITS can be left out).

printf: format specifiers

Specify how to print various types of things

```
printf "This phrase contains %d words\n" $a
This phrase contains 2 words

printf "This phrase contains %3d words\n" $a
This phrase contains 002 words

printf "This phrase contains %3.0d words\n" $a
This phrase contains   2 words

printf "This phrase contains %.4d words\n" $a
This phrase contains 0002 words

printf "This phrase contains %3.4d words\n" $a
This phrase contains 0002 words
```

printf: format specifiers

Specify how to print various types of things

%d Print the associated argument as **signed decimal** number
%i Same as %d
%o Print the associated argument as **unsigned octal** number
%u Print the associated argument as **unsigned decimal** number
%x Print the associated argument as **unsigned hexadecimal** number with lower-case hex-digits (a-f)
%X Same as %x, but with upper-case hex-digits (A-F)
%f Interpret and print the associated argument as **floating point** number
%e Interpret the associated argument as **double**, and print it in <N>±e<N> format
%E Same as %e, but with an upper-case E in the printed format

printf: format specifiers

Specify how to print various types of things

%g Interprets the associated argument as **double**, but prints it like %f or %e
%G Same as %g, but print it like %E
%c Interprets the associated argument as character: only the first character of a given argument is printed
%s Interprets the associated argument literally as string
%b Interprets the associated argument as a string and interpreting escape sequences in it
%q Prints the associated argument in a format, that it can be re-used as shell-input (escaped spaces etc..)

printf: format specifiers

Modifiers are specified between the introducting % and the character that specifies/identifies the format:

<N> **Any number:** specifies a **minimum field width**, if the text to print is smaller, it's padded with spaces
* **The asterisk:** the width is given as argument before the string. Usage (the "*" corresponds to the "20"): printf "%*s \n" 20 "test string"
# "Alternative format" for numbers: see table below
- **Left-bound** text printing into the field (standard is **right-bound**)
0 Pads numbers with zeros, not spaces
<space> Pad a positive number with a space, where a minus (-) is for negative numbers
+ Prints all numbers **signed** (+ for positive, - for negative)

printf: format specifiers

Precision for a floating- or double - precision number can be specified by using .<DIGITS>, where <DIGITS> is the number of digits for precision after the decimal point.

```
malleys-imac-2:ESCI7205 smalley$ printf "%.10f\n" 14.3
14.3000000000
```

Combine with <N> (total # characters, "-", decimal point, e)

```
smalleys-imac-2:ESCI7205 smalley$ printf "%15.10f\n" 14.3
   14.3000000000
```

printf: format specifiers

If <N> or <DIGITS> is an asterisk (*), the precision is read from the argument that precedes the number to print.

```
smalleys-imac-2:ESCI7205 smalley$ printf "%.*f\n" 10 4.3
4.3000000000
smalleys-imac-2:ESCI7205 smalley$ printf "%.*f\n" 10 14.3
14.3000000000
smalleys-imac-2:ESCI7205 smalley$ printf "%*.*f\n" 15 10 14.3
   14.3000000000
```

printf: format specifiers

Scientific notation
(seems to ignore the <N> field if too small)

```
smalleys-imac-2:~ smalley$ printf "%*.*e\n" 6 4 143200000000
1.4320e+11
smalleys-imac-2:~ smalley$ printf "%*.*e\n" 6 4 -143.200000000
-1.4320e+02
smalleys-imac-2:~ smalley$ printf "%*.*e\n" 6 3 -143.200000000
-1.432e+02
smalleys-imac-2:~ smalley$ printf "%.*e\n" 3 -143.200000000
-1.432e+02
smalleys-imac-2:~ smalley$printf "%*.*e\n" 15 3 -143.200000000
      -1.432e+02
smalleys-imac-2:~ smalley$
```

Later on we will talk about how computers represent numbers

Integer format
(integers, counting numbers)

Floating point format
(numbers with decimal point)

Floating point numbers can be
Real or Complex

printf: format specifiers

For strings, the precision specifies the maximum number of characters to print (i.e. the maximum field width).

(We already saw) For integers, it specifies the number of characters/digits to print (with leading zero- or blank-padding!).

Escape codes

\\ Prints the character \ (backslash)
\a Prints the alert character (ASCII code 7 decimal)
\b Prints a backspace
\f Prints a form-feed
\n Prints a newline
\r Prints a carriage-return
\t Prints a horizontal tabulator
\v Prints a vertical tabulator
\<NNN> Interprets <NNN> as octal number and prints the corresponding character from the character set
\0<NNN> same as \<NNN> *
\x<NNN> Interprets <NNN> as hexadecimal number and prints the corresponding character from the character set (3 digits)*
\u<NNNN> same as \x<NNN>, but 4 digits *
\U<NNNNNNNN> same as \x<NNN>, but 8 digits *

(* - indicates nonstandard, may not work)

A few of the most useful format specifiers

%s String
%c ASCII character. Print the first character of the corresponding argument
%d Decimal integer
%f Floating-point format
%E Scientific notation floating-point format

Special shell variables

$< : special BSD Unix csh command that essentially acts as read except it is not white space delimited

```
set name = "$<"
```

instead of

```
read firstname lastname
```

Special shell variables

$# : the number of arguments passed to the shell.

Useful when checking calling arguments (did you enter the correct number of them?), writing if:then:else blocks and loops.

We will cover this more later.

Special shell variables

"$@": (need quotes) represents all command line arguments at once, maintaining separation, same as "$1" "$2" "$3"

"$*" : (should have quotes) represents all command line arguments as one, same as "$1 $2 $3 $4"

Without quotes, $* equals "$@"

```
smalleys-imac-2:ESCI7205 smalley$ arglist.sh first second\
third

Listing args with "$@":
Arg #1 = first
Arg #2 = second
Arg #3 = third
Arg list seen as separate words.

Listing args with "$*":
Arg #1 = first second third
Entire arg list seen as single word.

Listing args with $* (unquoted):
Arg #1 = first
Arg #2 = second
Arg #3 = third
Arg list seen as separate words.
smalleys-imac-2:ESCI7205 smalley$
```

Special shell variables

$- : Options given to shell on invocation.

$? : Exit status of previous command.

$$: Process ID of shell process.

$! : Process ID of last background command.
Use this to save process ID numbers
for later use with the wait command.

Special shell variables

$IFS : Internal field separator
the list of characters that act as word separators.
Normally set to space and newline (maybe tab) (is a bash, not tcsh variable).

```
$ echo $IFS

$ echo $IFS | od -x
0000000 0a00
0000001
$ echo $IFS | od -c
0000000  \n
0000001
$
```

Special files

/dev/null : null device is a special file that discards all data written to it (but reports that the write operation succeeded), and provides no data to any process that reads from it (yielding EOF immediately).

Also know as the bit bucket, black hole, or a WOM (write only memory).

Special files
/dev/tty : redirects the script's standard-in to the terminal

Script

```
#!/bin/sh
printf "Hello.  My name is hdmacpro.  What is yours?\n"
read name < /dev/tty
printf "Nice to meet you %s.\n" $name
printf "Hello.  My name is hdmacpro.  What is yours?\n?"
read name
printf "Nice to meet you %s.\n?" $name
```

Run it

```
alpaca.ceri.memphis.edu517:> tsttty.sh
Hello.  My name is hdmacpro.  What is yours?
bob
Nice to meet you bob.
Hello.  My name is hdmacpro.  What is yours?
?Bob
Nice to meet you Bob.
?alpaca.ceri.memphis.edu518:>
```


Intro - arithmetic
SHELL SCRIPTING

Arithmetic

bash shell arithmetic resembles C programming language arithmetic

(very helpful if you don't already know C!).

In bash, the syntax $(( )) can be used to calculate arithmetic expressions or to set variables to complex arithmetic expressions

```
%echo $((3+4))
7
%echo $((x=2))
2
%echo $((++x))
3
%echo $((x++))
3
```

```
%echo $x
4
%((y=10))
%echo $y
10
```

Basic Arithmetic Operators

shell arithmetic is integer only

+ : addition
- : subtraction
* : multiplication
/ : division
% : remainder or modulus

```
%echo $(( 10%3 ))
1
%echo $(( 10/3 ))
3
```

Assignment Operators

= : set variable equal to value on right
(no spaces allowed around equals sign)

```
%x=2; echo $x
2
```

+= : set variable equal to itself plus the value on right (spaces allowed, but not required)

```
%x=2; echo $(( x +=2 ))
4
```

-= : set variable equal to itself minus the value on right (spaces allowed, but not required)

```
%x = 2; echo $((x-=2))
0
```

Assignment Operators

*= : set variable equal to itself times the value on right (spaces allowed, but not required).

```
%x = 2; echo $((x *= 4))
8
```

Assignment Operators

/= : set variable equal to itself divided by value on right (spaces allowed, but not required).

```
%x = 2; echo $((x/= 2))
1
```

%= : set variable equal to the remainder of itself divided by the value on the right

```
%x = 4; echo $((x %= 3))
1
```

Unary Operations

A unary expression contains one operand and one operator.

++ : increment the operand by 1

Unary Operations

if ++ occurs after the operand, $x++, the original value of the operand is used in the expression and then incremented.

if ++ occurs before the operand, ++$x, the incremented value of the operand is used in the expression.

Unary Operations

-- : decrement the operand by 1

+ : unary plus maintains the value of the operand, x=+x

-: unary minus negates the value of the operand, -1*x=-x

-! : logical negation

Intro – relational and logical operators, test

SHELL SCRIPTING

Relational Operators (in arithmetic expressions $((. . .)))

Returns 1 if true and 0 if false

All relational operators are left to right associative

= or == : test for equal to
< : test for less than
<= : test for less than or equal to
> : test for greater than
>= : test for greater than or equal to
!= : test for not equal

Bash does not understand floating point arithmetic.

It treats numbers containing a decimal point as strings.

Boolean (Logical) Operators

Boolean operators return 1 for true and 0 for false

&& : logical AND

tests that both expressions are true left to right associative

```
%echo $(( (3 < 4) && (10<15) ))
1
%echo $(( (3<4) && (10>15) ))
0
```

|| : logical OR

tests that one or both of the expressions are true left to right associative.

```
%echo $(( (3<4) || (10>15) ))
1
```

! : logical negation

tests negation of expression.

Bitwise Operators

Bitwise Operators treat operands as 16 (actually depends on word size on computer) bit binary values

Example: 4019 equals 0000111110110011_{base2} ($0FB3_{16}$ in hexadecimal) in integer format.

(Internally in the computer, integers are expressed in a format called two's-complement. Positive integers are in straight base 2. Negative integers are "funny".)

Bitwise Operators

~ : bitwise negation changes 0's to 1's (bits) and vice versa

& : bitwise AND

^ : bitwise exclusive OR

| : bitwise OR

<< : bitwise left shift (numerically is *2)

<<=n : bitwise left shift by n bits (numerically is $*2^n$)

>> : bitwise right shift (numerically is $\div 2^n$)

<<=n : bitwise left shift by n bits (numerically is $\div 2^n$)

Relational Operators (between character strings)

Returns 1 if true and 0 if false

All relational operators are left to right associative.

= or == : test for equal to
< : test for less than
> : test for greater than
!= : test for not equal

Relational Operators (between numerical values)

Returns 1 if true and 0 if false

All relational operators are left to right associative

-lt (<)
-gt (>)
-le (<=)
-ge (>=)
-eq (==)
-ne (!=)

test

test: condition evaluation utility

common scripting tool that tests expressions and many details about files using a long list of flags

Returns

0 if expression true and
1 if expression false or does not exist
(backwards to normal logic!)

test
two formats in bash scripting

test flag expression

or

[flag expression]

examples

```
bash-2.05$ [ 'abc' == 'abc' ]; echo $?
0
bash-2.05$ [ 'abc' = 'abc' ]; echo $?
0
bash-2.05$ [ "abc" != "def" ];echo $?
0
```

Note - we are testing character strings.

To test numerical values

```
$ test 3 -gt 4, echo $?
1
```

Note - the numerical tests are specified with a different format.

Returns

0 if expression true and
1 if expression false or does not exist
(backwards to normal logic!)

```
bash-2.05$ a=1
bash-2.05$ b=2
bash-2.05$ c=3
bash-2.05$ [ $a = 1 ];echo $?
0
bash-2.05$ [ $a -eq 1 ];echo $?
0
bash-2.05$ [ $a > 1 ];echo $?
0
bash-2.05$ [ $a -gt 1 ];echo $?
1
bash-2.05$ [ $b -eq 1 ];echo $?
1
bash-2.05$ [ $b -eq $c ];echo $?
1
bash-2.05$ [ $b -eq $(($c-1)) ];echo $?
0
bash-2.05$ [ $b == $(($c-1)) ];echo $?
0
bash-2.05$ [ $b == $(($c-2)) ];echo $?
1
```

What is this?

Test combinations with

-a (and) and -o (or)

```
if [ $# -eq 0 -o $# -ge 3 ]
then
. . .
fi

if [ \( $REGPARM = spat -o $REGPARM = chile \) -a $CMT = 1 ]
then
. . .
fi
```

(the [. . .]'s above are a form of the test expression)
(the backslashes are needed to "escape" the parentheses in the test expression)

You can use the return values together with && and || using the two test constructs

examples

```
$ test 3 -gt 4 && echo True || echo false
false
bash-2.05$[ $a = 1 ]&&[ $b == $(($c-1)) ];echo $?
0
bash-2.05$[ $a = 1 ]&&[ $b == $(($c-1)) ]&&[ $b -eq $c ];echo $?
1
bash-2.05$[ $a = 1 ]&&[ $b == $(($c-1)) ]||[ $b -eq $c ];echo $?
0
bash-2.05$[ $a = 1 ]&&( [ $b == $(($c-1)) ]||[ $b -eq $c ] );echo $?
0
bash-2.05$[ $a = 1 ]||( [ $b == $(($c-1)) ]&&[ $b -eq $c ] );echo $?
0
```

Some tests

-d Directory
-e Exists (also -a)
-f Regular file
-h Symbolic link (also -L)

(remember 0 is TRUE and 1 if FALSE!!!)

```
bash-2.05$ [ -e 'eqs.vim' ]; echo $?
0
bash-2.05$ [ -e 'eqs' ]; echo $?
1
bash-2.05$ filename=eqs.vim
bash-2.05$ echo $filename
eqs.vim
bash-2.05$ [ -e $filename ]; echo $?
0
bash-2.05$ test -d "$HOME" ;echo $?
0
```

Loops and Logic

SHELL SCRIPTING

do

. . .

Done

Does the commands in the "block" between do and done.

in bash, this construct is used in conjunction with loop structures for, while, and until

A 'for loop' is a programming language statement which allows code to be repeatedly executed.

```
for VARIABLE in 1 2 3 4 5 .. N
do
. . .
done
```

example

```
list=`ls -1 z*xyz`
for ITEM in $list
do
 #echo plot contour $ITEM
 psxy -R$REGION -$PROJ$SCALE -M$ -W5/$VLTGRAY $CONTINUE \
    $ITEM $VBSE >> $OUTPUTFILE
done
```

More examples

```
for i in 1 2 3 4 5
do
    echo "Welcome $i times"
done

for i in $(seq 1 2 20)
do
    echo "Welcome $i times"
done

for (( c=1; c<=5; c++ ))
do
   echo "Welcome $c times..."
done

for (( ; ; ))
do
    echo "infinite loops [ hit CTRL+C to stop]"
done
```

while:
continues to loop as long as the condition tests true

```
#!/bin/bash
. . .
while read vari1 vari2 … varin
do
    . . .
done < inputfile
```

This will read from the input file till it hits EOF (read returns 0, true, if there were no errors, on EOF [or an error] it returns a non zero value - false)

Full example

Script

```
#!/bin/bash
cat<<EOF>cities.dat
105.87 21.02 Hanoi LM
282.95 -12.1 LIMA LM
178.42 -18.13 SUVA LM
EOF

while read clon clat city junk
do
    echo $city $clon $clat
done < cities.dat
```

Run it

```
alpaca.ceri.memphis.edu516:> junk.sh
Hanoi 105.87 21.02
LIMA 282.95 -12.1
SUVA 178.42 -18.13
alpaca.ceri.memphis.edu517:
```

This script first makes the input data file, then reads it and prints out a part of it. Notice where the redirected input is located - at the end of the "command".

The structure of the while loop

While the test is true, do the block of code between the "do" and "done"

```
while test
do

    . . . block of code . . .

done
```

The structure of the while loop

The redirected input goes at the end.

One can enter the while command from the command line
(there is nothing special about it as far as the shell is concerned
(also notice where the semicolons, that separate lines, go).

```
%while read line; do echo "$line \n"; done < cities.dat
105.87 21.02 Hanoi LM \n
282.95 -12.1 LIMA LM \n
178.42 -18.13 SUVA LM \n
%
```

until:

until continues to loop as long as the condition exits unsuccessfully (is false)

(the until loop is used much less than the while loop)

```
#!/bin/bash
myvar=0
until [ $myvar –eq 5 ]     #until this expression is false
do
echo $myvar
myvar=$(( $myvar + 1 ))
done
```

```
%sh –f junk.sh
0
1
2
3
4
```

Break:

allows you to break out of a loop

can be used with a number to specify what do loop to break out of

```
while condition1        # Outer loop, loop 2
do
…
while condition2        # Inner loop, loop 1
do
…
break 2                 # Break out of outer loop (usually after
                                                  some test)
done
done
...                     # Execution continues here after break
```

if/then/fi

If the test is true, then run the block of code between the then and the fi (if spelled backwards).

```
if    [ $1 = "Heather" ]
then
    printf "Hi %s.  We were expecting you.\n" $1
fi
```

if/then/else/fi

If the test is true, run block of code between then and else. If the test is false, run block of code between else and fi.

```
if   [ $1 = "Heather" ]
then
   printf "Hi %s.  We were expecting you.\n" $1
else
   printf "Hi %s.  Nice to meet you.\n" $1
fi
```

if/then/elif/else/fi

If [test] is true, run block of code between then and elif. If it was false, do next [test]. If true, run block of code between else and elif. If false, do next [test], etc., or, finally (everything false to here) do block of code between else and fi.

```
if   [ $1 = "Heather" ]
then
   printf "Hi %s.  We were expecting you.\n" $1
elif [ $1 = "Andy"  ]
   printf "Hi %s.  We were expecting you too.\n" $1
elif [ $1 = "Gregg"  ]
   printf "Hi %s.  We were expecting you too.\n" $1
else
   printf "Hi %s.  Nice to meet you.\n" $1
fi
```

Can have logical combination of [tests]

```
if    [ $1 = "Heather" ] || [ $1 = "Andy" ]
then
    printf "Hi %s.  We were expecting you.\n" $1
elif [ $1 = "Andy"  ]
    printf "Hi %s.  We were expecting you too.\n" $1
else
    printf "Hi %s.  Nice to meet you.\n" $1
fi
```

```
if   [ $1 = "Heather" ] || [ $1 = "Andy" ]
then
    printf "Hi %s.  We were expecting you.\n" $1
elif [ $1 = "Andy"  ]
    printf "Hi %s.  We were expecting you too.\n" $1
else
    printf "Hi %s.  Nice to meet you.\n" $1
fi
```

Would this script ever output "We were expecting you too"?

(i.e. what is wrong with it?)

The case statement is an elegant replacement for if/then/else if/else statements when making numerous comparisons.

This recipe describes the case statement syntax for the Bourne family of shells

```
case "$var" in
value1)
commands;
;;
value2)
commands;
;;
*)
commands;
;;
esac
```

```
case "$var" in
value1)
commands;
;;
*)
commands;
;;
esac
```

The case statement compares the value of the variable ($var in this example) to one or more values (value1, value2, …).
Once a match is found, the associated commands are executed and the case statement is terminated.
The optional last comparison "*)" is a default case and will match anything.

For example, branching on a command line parameter to the script, such as 'start' or 'stop' with a runtime control script.

The following example uses the first command line parameter ($1):

```
case "$1" in
'start')
/usr/app/startup-script
;;
'stop')
/usr/app/shutdown-script
;;
'restart')
echo "Usage: $0 [start|stop]"
;;
esac
```

Some tcsh/csh syntax

A shell with C language-like syntax.

Control structures

- foreach, if, switch and while

foreach : a tcsh command is a powerful way to iterate over files from the tcsh command line (can also put in shell scripts - don't get prompts).

```
%foreach file ( 828/*BHZ* )#set variable file to each sac file
foreach? echo $file
foreach? set name = `echo $file | cut -f2 -d'/' `
foreach? set sta = `echo $name | cut -f1 -d'.' `
foreach? echo "copy $file to $sta.BHZ.SAC
foreach? cp $file $sta.BHZ.SAC
foreach? end
828/GAR.BHZ_00.D.1989.214:10.24.59
copy 828/GAR.BHZ_00.D.1989.214:10.24.59 to GAR.BHZ.SAC
```

Aside - new command

cut

The cut command has the ability to cut out characters or fields. cut uses delimiters.

```
file = 828/GAR.BHZ_00.D.1989.214:10.24.59
Set name = `echo $file | cut -f2 -d'/'`
```

Says return the second field (-f2), using '/' as a delimiter (-d'/') (assign it to the variable name)

```
name = GAR.BHZ_00.D.1989.214:10.24.59
set sta = `echo $name | cut -f1 -d'.' `
```

Says return the first field (-f1), using '.' as a delimiter (-d'.') (assign it to the variable sta)

```
sta = GAR
```

If-then-else block in tcsh/csh

Two formats

```
if (expression) simple command
```

or

```
if (expression) then
   ...
else
   ...
endif
```

The tcsh/csh switch statement can replace several if ... then statements.

```
switch (string)
  case pattern1:
    commands...
    breaksw
  case pattern2:
    commands...
    breaksw
  default:
    commands...
    breaksw
endsw
```

For the string given in the switch statement's argument, commands following the case statement with the matching pattern are executed until the endsw statement.

These patterns may contain ? and * to match groups of characters or specific characters.

switch/case in tcsh syntax

```
foreach plane(0035.0 0050.0)
set cnt=`expr $cnt + 1`
switch ($cnt)
    case 1:
        set xpos=-5.
        set ypos=4.75
        set min=-2.5
        set max=2.5
        breaksw
    case 2:
        set xpos=-6.6
        set ypos=-3.5
        set min=2.5
        set max=7
        breaksw
    endsw
. . . such as excessive amounts of GMT
end
```

Another example

```
# Get the arguments
set source_dir = $1
set target_dir = $2
shift argv
shift argv
while ($#argv > 0 )
    set input = ( $argv )
    switch($input[1])
        case -m:
            set module = $input[2]
            breaksw
        case -auto:
            set auto = 'Y'
            breaksw
        case -full:
            set full = 'Y'
            breaksw
    endsw
    shift argv
end
```

Built-in shell variables

argv Special variable used in shell scripts to hold the value of command line arguments.

Awk (lecture 1)

AWK PROGRAMMING LANGUAGE

Awk Programming Language

standard unix language that is geared for text processing and creating formatted reports

But is very valuable to seismologists because it uses floating point math, and is designed to work with columnar data

syntax similar to C and bash

one of the most useful unix tools at your command

Considers text files as having records (lines), which have fields (columns)

Performs floating & integer arithmetic and string operations

Has loops and conditionals

Can define your own functions (subroutines)

Can execute unix commands within the scripts and process the results

Versions/Implementations

awk: original awk

nawk: new awk, dates to 1987

gawk: GNU awk has more powerful string functionality

the CERI unix system has all three. You want to use nawk. I suggest adding this line to your .cshrc file

```
alias awk 'nawk'
```

in OS X, awk is already nawk so no changes are necessary

Command line functionality

you can call awk from the command line two ways:

```
awk [options] '{ commands }' variables infile(s)
awk –f scriptfile variables infile(s)
```

or you can create an executable awk script

```
%cat << EOF > test.awk
#!/usr/bin/nawk
some set of commands
EOF

%chmod 755 test.awk
%./test.awk
```

How awk treats text

awk commands are applied to every record or line of a file

it is designed to separate the data in each line into a number of fields

essentially, each field becomes a member of an array so that the first field is $1, second field $2 and so on.

$0 refers to the entire record.

Field Separator

the default field separator is one or more white spaces

```
$1  $2    $3 $4 $5 $6 $7      $8         $9        $10     $11
1  1918 9   22 9   54 49.29   -1.698    98.298    15.1   ehb
```

So $1 = 1, $2=1918, …, 410=15.1, $11=ebb

Notice that the fields may be integer, floating point (have a decimal point) or strings. Nawk is generally smart enough to figure out how to use them.

Field Separator

the field separator may be modified by resetting the FS built in variable

Look at passwd file

```
%head -n1 /etc/passwd
root:x:0:1:Super-User:/:/sbin/sh
```

Separator is ":", so reset it.

```
%awk –F":" '{ print $1, $3}'   /etc/passwd
root 0
```

Print

One of the most common awk commands

awk is not sensitive to white space in the commands

```
%awk –F":"   '{ print $1 $3}'   /etc/passwd
Root0
```

The two output fields (Root and 0) are run together - two solutions to this

```
%awk –F":"   '{ print $1 "   " $3}'   /etc/passwd
%awk –F":"   '{ print $1, $3}'   /etc/passwd
root 0
```

any string or numeric text can be explicitly output using “”

Assume our input file looks like this

```
1 1 1918 9 22 9 54 49.29  -1.698 98.298 15.0 0.0
0.0 ehb
```

```
1 1 1918 9 22 9 54 49.29  -1.698 98.298 15.0 0.0
0.0 ehb FEQ  x
```

```
%awk '{print "latitude:",$9,"longitude:",$10,"depth:",\
$11}'  SUMA.Loc
latitude: -1.698 longitude: 98.298 depth: 15.0
latitude: 9.599 longitude: 92.802 depth: 30.0
latitude: 4.003 longitude: 94.545 depth: 20.0
```

you can specify a newline in two ways

```
%awk '{print "latitude:",$9; print "longitude:",$10}' SUMA.Loc
%awk '{print "latitude:",$9"\n","longitude:",$10}'  SUMA. Loc
latitude: -1.698
longitude: 98.298
```

Assignment Operators

= : set variable equal to value on right

+= : set variable equal to itself plus the value on right (VOR)

-= : set variable equal to itself minus VOR

*= : set variable equal to itself times VOR

/= : set variable equal to itself divided by VOR

%= : set variable equal to the remainder of itself divided by VOR

^= : set variable equal to itself to the power/exponent following the equal sign

Unary Operations

A unary expression contains one operand and one operator

++ : increment the operand by 1

if ++ occurs after, $x++, the original value of the operand is used in the expression and then incremented.

if ++ occurs before, ++$x, the incremented value of the operand is used in the expression.

Unary Operations

-- : decrement the operand by 1

+ : unary plus maintains the value of the operand, x=+x

-: unary minus negates the value of the operand, -1*x=-x

-! : logical negation evaluates if the operand is true (returns 1) or false (returns 0)