CSCI 330
The UNIX System

Bash Programming

Shells

- A shell can be used in one of two ways:
  - A *command interpreter*, used interactively
  - A *programming language*, to write shell scripts (your own custom commands)

Basic Shell Programming

- A shell script is just a file containing shell commands, but with a few extras:
  - data structure: variables
  - control structure: sequence, decision, loop
- The first line of a shell script (Shebang) should be a comment of the following form:

```
#! /bin/bash
#! /bin/sh
```

Bash scripting and Shell scritpting

- not all **bash commands** will work in sh, but most **shell commands** will work in bash. In general, most users will want to use /bin/bash for their scripts so they can take advantage of the extended feature set; system scripts can be executed with /bin/sh if it is required.
- A shell script must be readable and executable. a shell script has to be “in your path” to be executed:
  - make executable: `% chmod u+rx scriptname`
  - Run the script: `% ./scriptname`

Bash shell programming

- Input
  - prompting user
  - command line arguments
- Decision:
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for
  - select
- Functions
- Traps

User input

- shell allows to prompt for user input

Syntax:

```
read varname [more vars]
```

- or

```
read –p "prompt" varname [more vars]
```

- words entered by user are assigned to **varname** and "**more vars**"
- Leftover input words are all assigned to the last variable

User input example First Way

```
#! /bin/sh
read -p "enter your name: " first last

echo "First name: $first"
echo "Last name: $last"
```

- The user variable name can be any sequence of letters, digits, and the underscore character, but the first character must be a letter.
- To assign a value to a variable:

```
number=25

name="Bill Gates"
```

- There cannot be any space before or after the “=“
- Internally, all values are stored as strings.
- To use a variable, precede the name with a “$”:

User input example Second Way

```
$ cat test2
#!/bin/sh
echo "Enter name: "
read name
echo "How many friends do you have? "
read number
echo "$name has $number friends!"
$ test2
Enter name:
Bill Gates
How many friends do you have?
too many
Bill Gates has too many friends!
```

Examples: Command Line Arguments

```
% set tim bill ann fred
      $1  $2   $3  $4
% echo $*
tim bill ann fred
% echo $#
4
% echo $1
tim
% echo $3 $4
ann fred
```

The ‘set’ command can be used to assign values to positional parameters

Leftover input words are all assigned to the last variable

- "read" reads one line of input from the keyboard and assigns it to one or more user-supplied variables.

```
$ cat test3
#!/bin/sh
echo "Enter name and how many friends:"
read name number
echo "$name has $number friends!"
$ test3
Enter name and how many friends:
Bill Gates 63
Bill has Gates 63 friends!
$ test3
Enter name and how many friends:
Bill
Bill has  friends!
```

Example

```
$ cat test1
  #!/bin/sh
  number=25
  name="Bill Gates"
  echo "$number $name"
  $ test1
  25 Bill Gates
```

- Use a backslash before $ if you really want to print the dollar sign:

```
$ cat test4
#!/bin/sh
    echo "Enter amount: "
    read cost
    echo "The total is: \$$cost"
    $ test4
    Enter amount:
    18.50
    The total is $18.50
```

- You can also use single quotes for printing dollar signs.
- Single quotes turn off the special meaning of all enclosed dollar signs:

```
$ cat test5
#!/bin/sh
    echo "Enter amount: "
    read cost
    echo 'The total is: $' "$cost"
    $ test5
    Enter amount:
    18.50
    The total is $ 18.50
```

Special shell variables

Parameter	Meaning
$0	Name of the current shell script
$1-$9	Positional parameters 1 through 9
$#	The number of positional parameters
$*	All positional parameters, “$*” is one string
$@	All positional parameters, “$@” is a set of strings
$?	Return status of most recently executed command
$$	Process id of current process

expr

- Shell programming is not good at numerical computation, it is good at text processing.
- However, the `expr` command allows simple integer calculations.
- Here is an interactive Bourne shell example:

```
$ i=1
$ expr $i + 1
2
```

- To assign the result of an `expr` command to another shell variable, surround it with backquotes:

```
$ i=1
$ i=`expr $i + 1`
$ echo "$i"
2
```

expr

- The * character normally means “all the files in the current directory”, so you need a “\” to use it for multiplication:

```
$ i=2
$ i=`expr $i \* 3`
$ echo $i
6
```

- `expr` also allows you to group expressions, but the “(“ and “)” characters also need to be preceded by backslashes:

```
$ i=2
$ echo `expr 5 + \( $i \* 3 \)`
11
```

example

```
$ cat test6
#!/bin/sh
        echo "Enter height of rectangle: "
        read height
        echo "Enter width of rectangle: "
        read width
        area=`expr $height \* $width`
        echo "The area of the rectangle is $area"
        $ test6
        Enter height of rectangle:
        10
        Enter width of rectangle:
        5
        The area of the ractangle is 50
        $ test6
        Enter height of rectangle:
        10.1
        Enter width of rectangle:
        5.1
        expr: non-numeric argument
```


Does not work for floats!

Command Substitution

- A command or pipeline surrounded by backquotes causes the shell to:
  - Run the command/pipeline
  - Substitute the output of the command/pipeline for everything inside the quotes
- You can use backquotes anywhere:

```
$ cat test7
#!/bin/sh
    user=`whoami`
    numusers=`who | wc -l`
    echo "Hi $user! There are $numusers users
    logged on."
    $ test7
    Hi gates! There are      6 users logged on.
```

bash control structures

- if-then-else
- case
- loops
  - for
  - while
  - until
  - select

if statement

```
if command
then
  statements
fi
```

- statements are executed only if **command** succeeds, i.e. has return status “0”

test command

Syntax:

```
test expression
[ expression ]
```

- evaluates ‘expression’ and returns true or false

Example:

```
if test -w "$1"
  then
  echo "file $1 is write-able"
fi
```

The simple if statement

```
if [ condition ]; then
  statements
fi
```

- executes the statements only if **`condition`** is true

```
$ cat test7
#!/bin/sh
    user=`whoami`
    if [ $user = "clinton" ]
    then
        echo "Hi Bill!"
    fi
    $ test7
    Hi Bill!
```

- the spaces before and after the square brackets [ ] are required.

The if-then-else statement

```
if [ condition ]; then
   statements-1
else
   statements-2
fi
```

- executes statements-1 if condition is true
- executes statements-2 if condition is false

example

```
$ cat test7
#!/bin/sh
    user=`whoami`
    if [ $user = "clinton" ]
    then
        echo "Hi Bill!"
    else
        echo "Hi $user!"
    fi
    $ test7
    Hi horner!
```

The if…statement

```
if [ condition ]; then
    statements
elif [ condition ]; then
    statement
else
    statements
fi
```

- The word `elif` stands for “else if”
- It is part of the if statement and cannot be used by itself

Example

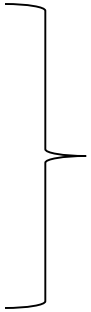
```
$ cat test8
#!/bin/sh
     users=`who | wc -l`
     if [ $users -ge 4 ]
     then
          echo "Heavy load"
     elif [ $users -gt 1 ]
     then
          echo "Medium load"
     else
          echo "Just me!"
     fi
     $ test8
     Heavy load!
```

Relational Operators

Meaning	Numeric	String
Greater than	-gt	
Greater than or equal	-ge	
Less than	-lt	
Less than or equal	-le	
Equal	-eg	= or ==
Not equal	-ne	!=
str1 is less than str2		str1 < str2
str1 is greater str2		str1 > str2
String length is greater than zero		-n str
String length is zero		-z str

Compound logical expressions

!	not
&&	and
||	or



and, or
must be enclosed within

[[ ]]

Example: Using the ! Operator

```
#!/bin/bash

read -p "Enter years of work: " Years
if [ ! "$Years" -lt 20 ]; then
   echo "You can retire now."
else
   echo "You need 20+ years to retire"
fi
```

Example: Using the && Operator

```
#!/bin/bash

Bonus=500
read -p "Enter Status: " Status
read -p "Enter Shift: " Shift
if [[ "$Status" = "H" && "$Shift" = 3 ]]
then
   echo "shift $Shift gets \$$Bonus bonus"
else
   echo "only hourly workers in"
   echo "shift 3 get a bonus"
fi
```

Example: Using the || Operator

```
#!/bin/bash

read -p "Enter calls handled:" CHandle
read -p "Enter calls closed: " CClose
if [[ "$CHandle" -gt 150 || "$CClose" -gt 50 ]]
   then
   echo "You are entitled to a bonus"
else
   echo "You get a bonus if the calls"
   echo "handled exceeds 150 or"
   echo "calls closed exceeds 50"
fi
```

File Testing

	Meaning
-d file	True if ‘file’ is a directory
-f file	True if ‘file’ is an ord. file
-r file	True if ‘file’ is readable
-w file	True if ‘file’ is writable
-x file	True if ‘file’ is executable
-s file	True if length of ‘file’ is nonzero

Example: File Testing

```
#!/bin/bash
echo "Enter a filename: "
read filename
if [ ! -r "$filename" ]
 then
   echo "File is not read-able"
 exit 1
fi
```

Example: File Testing

```
#! /bin/bash

if [ $# -lt 1 ]; then
        echo "Usage: filetest filename"
        exit 1
fi
if [[ ! -f "$1" || ! -r "$1" || ! -w "$1" ]]
then
  echo "File $1 is not accessible"
  exit 1
fi
```

Example: if… Statement

```
# The following THREE if-conditions produce the same result

* DOUBLE SQUARE BRACKETS
read -p "Do you want to continue?" reply
if [[ $reply = "y" ]]; then
   echo "You entered " $reply
fi

* SINGLE SQUARE BRACKETS
read -p "Do you want to continue?" reply
if [ $reply = "y" ]; then
   echo "You entered " $reply
fi

* "TEST" COMMAND
read -p "Do you want to continue?" reply
if test $reply = "y"; then
   echo "You entered " $reply
fi
```

Example: if..elif... Statement

```
#!/bin/bash

read -p "Enter Income Amount: " Income
read -p "Enter Expenses Amount: " Expense

let Net=$Income-$Expense

if [ "$Net" -eq "0" ]; then
   echo "Income and Expenses are equal - breakeven."
elif [ "$Net" -gt "0" ]; then
   echo "Profit of: " $Net
else
   echo "Loss of: " $Net
fi
```

The case Statement

- use the case statement for a decision that is based on multiple choices

Syntax:

```
case word in
 pattern1) command-list1
 ;;
 pattern2) command-list2
 ;;
 patternN) command-listN
 ;;
esac
```

case pattern

- checked against word for match
- may also contain:

  `*`

  `?`

  `[ … ]`

  `[:class:]`
- multiple patterns can be listed via:

  `|`

Example 1: The case Statement

```
#!/bin/bash
echo "Enter Y to see all files including hidden files"
echo "Enter N to see all non-hidden files"
echo "Enter q to quit"

read -p "Enter your choice: " reply

case $reply in
  Y|YES) echo "Displaying all (really…) files"
         ls -a ;;
  N|NO)  echo "Display all non-hidden files..."
         ls ;;
  Q)     exit 0 ;;

  *) echo "Invalid choice!"; exit 1 ;;
esac
```

Example 2: The case Statement

```
#!/bin/bash
ChildRate=3
AdultRate=10
SeniorRate=7
read -p "Enter your age: " age
case $age in
  [1-9]|[1][0-2])    # child, if age 12 and younger
     echo "your rate is" '$'"$ChildRate.00" ;;
    # adult, if age is between 13 and 59 inclusive
  [1][3-9]|[2-5][0-9])
     echo "your rate is" '$'"$AdultRate.00" ;;
  [6-9][0-9])        # senior, if age is 60+
     echo "your rate is" '$'"$SeniorRate.00" ;;
esac
```

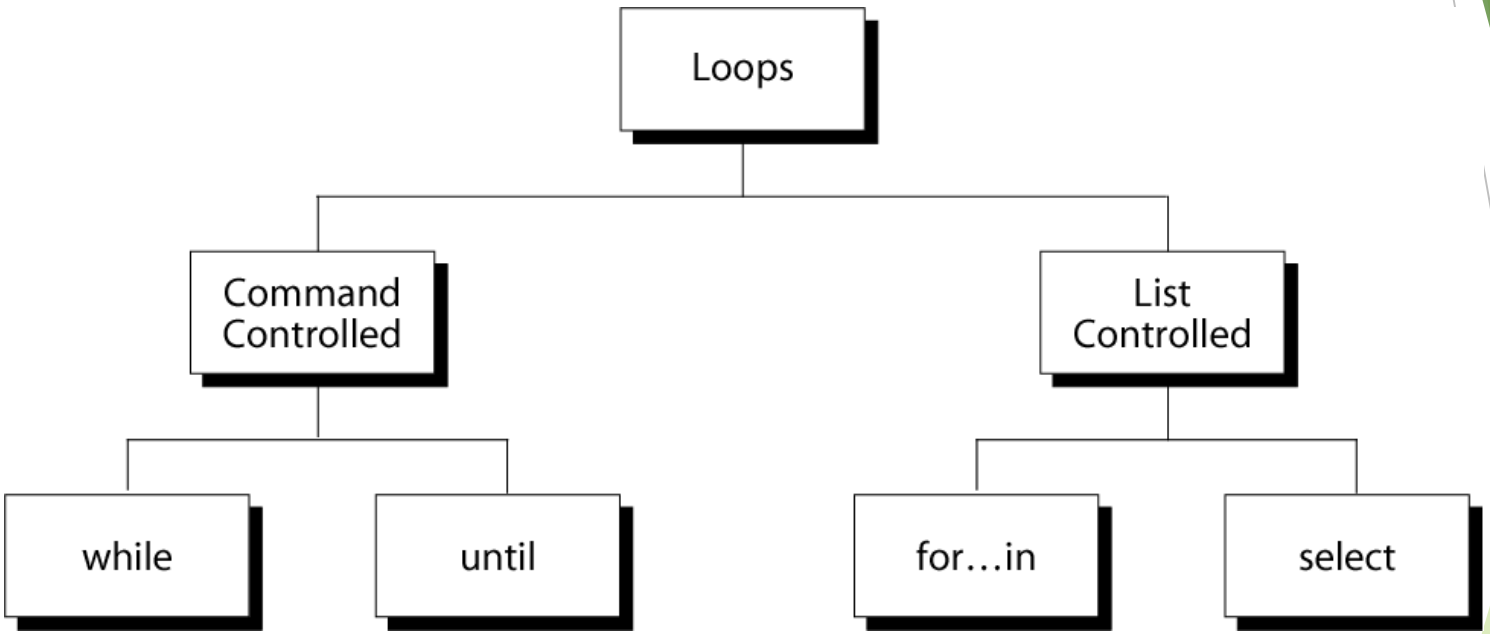
Bash programming: so far

- Data structure
  - Variables
  - Numeric variables
  - Arrays
- User input
- Control structures
  - if-then-else
  - case

Bash programming: still to come

- Control structures
  - Repetition
    - do-while, repeat-until
    - for
    - select
- Functions
- Trapping signals

Repetition Constructs



The while Loop

- Purpose:

  To execute commands in “command-list” as long as “expression” evaluates to true

Syntax:

```
while [ expression ]
do
 command-list
done
```

Example: Using the while Loop

```
#!/bin/bash
COUNTER=0
while [ $COUNTER -lt 10 ]
do
    echo The counter is $COUNTER
    let COUNTER=$COUNTER+1
done
```

Example: Using the while Loop

```
#!/bin/bash

Cont="Y"
while [ $Cont = "Y" ]; do
  ps -A
  read -p "want to continue? (Y/N)" reply
  Cont=`echo $reply | tr [:lower:]
   [:upper:]`
done
echo "done"
```

Example: Using the while Loop

```
#!/bin/bash
# copies files from home- into the webserver- directory
# A new directory is created every hour

PICSDIR=/home/carol/pics
WEBDIR=/var/www/carol/webcam
while true; do
   DATE=`date +%Y%m%d`
   HOUR=`date +%H`
   mkdir $WEBDIR/"$DATE"
   while [ $HOUR -ne "00" ]; do
      DESTDIR=$WEBDIR/"$DATE"/"$HOUR"
      mkdir "$DESTDIR"
      mv $PICSDIR/*.jpg "$DESTDIR"/
      sleep 3600
      HOUR=`date +%H`
   done
done
```

The until Loop

- Purpose:

  To execute commands in “command-list” as long as “expression” evaluates to false

Syntax:

```
until [ expression ]
do
 command-list
done
```

Example: Using the until Loop

```
#!/bin/bash

COUNTER=20
until [ $COUNTER -lt 10 ]
do
    echo $COUNTER
    let COUNTER-=1
done
```

Example: Using the until Loop

```
#!/bin/bash

Stop="N"
until [ $Stop = "Y" ]; do
  ps -A
  read -p "want to stop? (Y/N)" reply
  Stop=`echo $reply | tr [:lower:]
   [:upper:]`
done
echo "done"
```

The for Loop

- Purpose:

  To execute commands as many times as the number of words in the “argument-list”

Syntax:

```
for variable in argument-list
do
 commands
done
```

Example 1: The for Loop

```
#!/bin/bash

for i in 7 9 2 3 4 5
do
   echo $i
done
```

Example 2: Using the for Loop

```
#!/bin/bash
# compute the average weekly temperature

for num in 1 2 3 4 5 6 7
do
   read -p "Enter temp for day $num: " Temp
   let TempTotal=$TempTotal+$Temp
done

let AvgTemp=$TempTotal/7
echo "Average temperature: " $AvgTemp
```

looping over arguments

- simplest form will iterate over all command line arguments:

```
#! /bin/bash
for parm
do
        echo $parm
done
```

Select command

- Constructs simple menu from word list
- Allows user to enter a number instead of a word
- User enters sequence number corresponding to the word

Syntax:

```
select WORD in LIST
do
 RESPECTIVE-COMMANDS
done
```

- Loops until end of input, i.e. ^d (or ^c)

Select example

```
#! /bin/bash
select var in alpha beta gamma
do
      echo $var
done
```

- Prints:

```
1) alpha
2) beta
3) gamma
#? 2
beta
#? 4
#? 1
alpha
```

Select detail

- PS3 is select sub-prompt
- $REPLY is user input (the number)

```
#! /bin/bash
PS3="select entry or ^D: "
select var in alpha beta
do
      echo "$REPLY = $var"
done
```

Output:

```
select ...
1) alpha
2) beta
? 2
2 = beta
? 1
1 = alpha
```

Select example

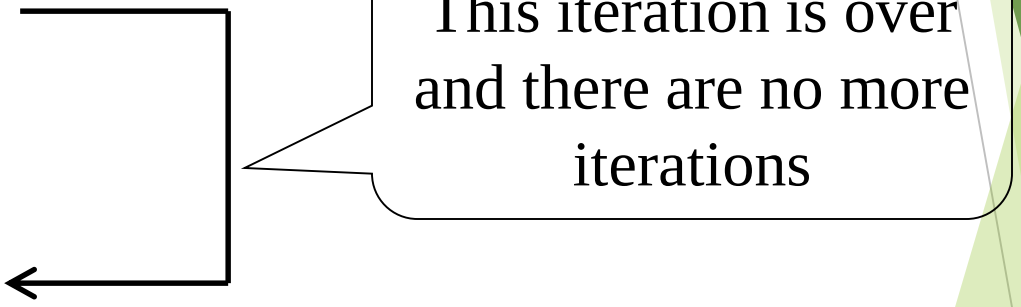
```
#!/bin/bash
echo "script to make files private"
echo "Select file to protect:"

select FILENAME in *
do
  echo "You picked $FILENAME ($REPLY)"
  chmod go-rwx "$FILENAME"
  echo "it is now private"
done
```

break and continue

- Interrupt for, while or until loop
- The break statement
  - transfer control to the statement AFTER the done statement
  - terminate execution of the loop
- The continue statement
  - transfer control to the statement TO the done statement
  - skip the test statements for the current iteration
  - continues execution of the loop

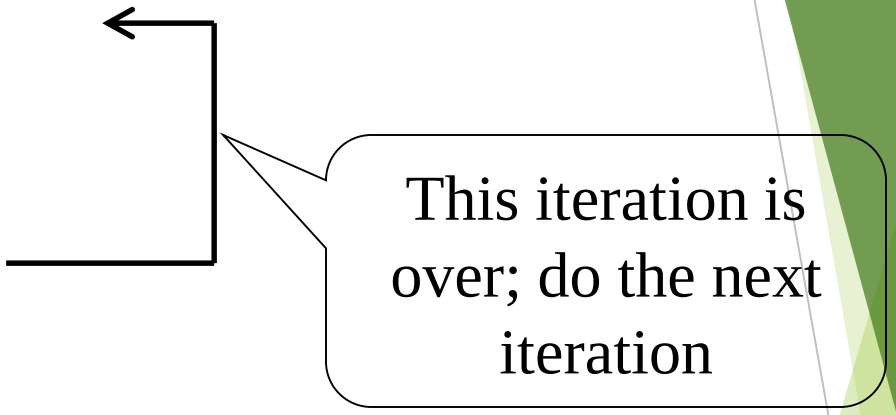
The break command

```
while [ condition ]
do
     cmd-1
     break
     cmd-n
done
echo "done"
```



The continue command

```
while [ condition ]
do
     cmd-1
     continue
     cmd-n
done
echo "done"
```



Example:

```
for index in 1 2 3 4 5 6 7 8 9 10
do
        if [ $index -le 3 ]; then
             echo "continue"
             continue
        fi
        echo $index
        if   [ $index -ge 8 ]; then
             echo "break"
             break
        fi
done
```

Bash shell programming

- Sequence
- Decision:
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for
  - select
- Functions
- Traps

DONE !

still to come

Shell Functions

- A shell function is similar to a shell script
  - stores a series of commands for execution later
  - shell stores functions in memory
  - shell executes a shell function in the same shell that called it
- Where to define
  - In .profile
  - In your script
  - Or on the command line
- Remove a function
  - Use unset built-in

Shell Functions

- must be defined before they can be referenced
- usually placed at the beginning of the script

Syntax:

```
function-name () {
    statements
}
```

Example: function

```
#!/bin/bash

funky () {
  # This is a simple function
  echo "This is a funky function."
  echo "Now exiting funky function."
}

# declaration must precede call:

funky
```

Example: function

```
#!/bin/bash
fun () { # A somewhat more complex function.
    JUST_A_SECOND=1
    let i=0
    REPEATS=30
    echo "And now the fun really begins."
    while [ $i -lt $REPEATS ]
    do
      echo "-------FUNCTIONS are fun-------->"
      sleep $JUST_A_SECOND
      let i+=1
    done
}
fun
```

Function parameters

- Need not be declared
- Arguments provided via function call are accessible inside function as $1, $2, $3, …

$# reflects number of parameters

$0 still contains name of script

(not name of function)

Example: function with parameter

```
#! /bin/sh
testfile() {
  if [ $# -gt 0 ]; then
     if [[ -f $1 && -r $1 ]]; then
        echo $1 is a readable file
     else
        echo $1 is not a readable file
     fi
  fi
}

testfile .
testfile funtest
```

Example: function with parameters

```
#! /bin/bash
checkfile() {
   for file
   do
      if [ -f "$file" ]; then
         echo "$file is a file"
      else
         if [ -d "$file" ]; then
            echo "$file is a directory"
         fi
      fi
   done
}
checkfile . funtest
```

Local Variables in Functions

- Variables defined within functions are global,

  i.e. their values are known throughout the entire shell program

- keyword “local” inside a function definition makes referenced variables “local” to that function

Example: function

```
#! /bin/bash

global="pretty good variable"

foo () {
        local inside="not so good variable"
        echo $global
        echo $inside
        global="better variable"
}

echo $global
foo
echo $global
echo $inside
```

Handling signals

- Unix allows you to send a signal to any process

- -1 = hangup `kill -HUP 1234`
- -2 = interrupt with ^C `kill -2 1235`
- no argument = terminate `kill 1235`
- -9 = kill `kill -9 1236`
  - -9 cannot be blocked

- list your processes with

```
ps -u userid
```

Signals on Linux

```
% kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

- ^C is 2 - SIGINT

Handling signals

- Default action for most signals is to end process
  - term: signal handler

- Bash allows to install custom signal handler

Syntax:

```
trap 'handler commands' signals
```

Example:

```
trap 'echo do not hangup'  1 2
```

Example: trap hangup

```
#! /bin/bash
# kill -1 won't kill this process
# kill -2 will

trap 'echo dont hang up' 1

while true
do
        echo "try to hang up"
        sleep 1
done
```

Example: trap multiple signals

```
#! /bin/sh
# plain kill or kill -9 will kill this
trap 'echo 1' 1
trap 'echo 2' 2

while true; do
   echo -n .
   sleep 1
done
```

Example: removing temp files

```
#! /bin/bash
trap 'cleanup; exit' 2

cleanup () {
        /bin/rm -f /tmp/tempfile.$$.?
}

for i in 1 2 3 4 5 6 7 8
do
        echo "$i.iteration"
        touch /tmp/tempfile.$$.$i
        sleep 1
done
cleanup
```

Restoring default handlers

- `trap` without a command list will remove a signal handler
- Use this to run a signal handler once only

```
#! /bin/sh
trap 'justonce' 2
justonce() {
  echo "not yet"
  trap 2              # now reset it
  }

while true; do
   echo -n "."
   sleep 1
done
```

Debug Shell Programs

- Debugging is troubleshooting errors that may occur during the execution of a program/script
- The following two commands can help you debug a bash shell script:
  - echo

    use explicit output statements to trace execution
  - set

Debugging using “set”

- The “set” command is a shell built-in command
- has options to allow flow of execution
  - -v option prints each line as it is read
  - -x option displays the command and its arguments
  - -n checks for syntax errors
- options can turned on or off
  - To turn on the option: set -xv
  - To turn off the options: set +xv
- Options can also be set via she-bang line

```
#! /bin/bash -xv
```

Summary: Bash shell programming

- Sequence
- Decision:
  - if-then-else
  - case
- Repetition
  - do-while, repeat-until
  - for
  - select
- Functions
- Traps

DONE !