System Programming

Basic Shell Scripting

Shell Script (Program)

- What is a shell script?
  - Shell commands in a text file that is invoked as its own command
- Commands include
  - anything you can type on the command line
  - shell variables
  - control statements (`if, while, for, ...`)

Resources

- Online
  - Advanced bash-scripting guide
    http://www.tldp.org/LDP/abs/html/index.html
  - ksh Reference Manual
    http://www.bolthole.com/solaris/ksh.html

Script Execution

- Provide script as an argument to a shell command – `bash my_script`
- Or specify shell on the first line of the script – `#!/bin/bash`
  - Make sure that the script is executable
  - Run `my_script` directly from the command line
- No compilation; interpreted by shell

Simple Script

```bash
#!/bin/bash
echo "Hello, World!"
path=$(pwd)
echo $path
```

Result:

```
Hello, World!
/home/user2
```

Shell Variables

- Numeric
- Strings
- Arrays
- Command line arguments
  - Read only
- Functions
- `var` refers to the name of the variable, $var to the value
  - `var=100` # sets the value to `100`
  - `echo ”\$var = $var”` # will print `$var = 100`
- Remove a variable with `unset var`
- Names begin with a letter and can include letters, digits, and underscore

Numeric Variables

- Integer variables are the only pure numeric variables that can be used in bash
- Declaration and setting value:

```
 declare -i var=100
```

- Numeric expressions are enclosed in double parentheses (in the style of C). General format:

```
   var=$((expression))  or  var=$[expression]
```

  - `e.g. i=$((var++))            or (( var+=1 ))`
  - `e.g. i=$(( var2=1+$var )) or ((var2=1+var))`
  - `e.g. i=$[var+2]              -- [var+=2] Error!`
  - `e.g. echo $((var*7))         or echo $[var*7]`
  - `e.g. echo ((var2=1+var))  or echo [var2=1+var]`
- Operators are the same as in C/C++
  - `+,-,*,/,%,&,|,<,>,<=,>=,==,!=,&&,||,`
    `+=,-=,*=,/=,%=,~,~=,!,<<,>>,^`

String Variables

- Unless explicitly declared as another type, variables are strings
- `var=100` makes the `var` the string "`100`".
- However, placing the variable within double parentheses will treat it as an integer
  - `(( var2=1+$var ))`

String Variables (cont.)

- Using substrings
  - ${string:*n*} # n: index
    - ${string:5} # except first five chars
    - ${string:(-2)} # last two chars
  - ${string:*n*:*m*} # n: index, m: number
    - ${string:0:5} # first to fifth (5 chars)
    - ${string:1:3} # second to fourth (3 chars)
  - ${#string} # length of string
- Concatenating strings
  - `var1="$var1 $var2"`
- Manipulating string
  - `y=${x:${#x}-1}${x:1:${#x}-2}${x:0:1}`

Variable Substitution

- $name or ${name} # use the value of name
- ${name:-value} # if name is not set, use value
- ${name:=value} # if name is not set, use value and assign it to name
- ${name:?value} # if name is not set, write value to stderr
- ${name:+value} # if name is set, use value; otherwise use null
- ${name%pattern} # remove smallest suffix pattern
- ${name%%pattern} # remove largest suffix pattern
- ${name#pattern} # remove smallest prefix pattern
- ${name##pattern} # remove largest prefix pattern
  - Filename substitution characters (*, ?, [...], !) may be used in pattern.

Array Variables

- Array is a list of values – do not have to declare size
- Reference a value by `${name[index]}`
  - `${a[3]}`   # value in fourth position
  - `$a`   # same as `${a[0]}`
- Use the `declare –a` command to declare an array
  - `declare –a sports`
  - `sports=(basketball football soccer)`
  - `sports[3]=hockey`

Array Variables (cont.)

- Array initialization
  - `sports=(football basketball)`
  - `moresports=($sports tennis)`
- `${array[@]}` or `${array[*]}` refers to the entire contents of the array
  - `echo ${moresports[*]}`
  - Output: `football tennis`
- `${#array[*]}` refers the number of values in the array
  - `echo ${#moresports[*]}`
  - Output: 2

Exported Variables

- The `export` command allows child processes of the shell to access the variable
  - `export <variables>`
  - `declare –x <vars>`
- `export –p` shows a list of the variables and their values exported by your shell

- A program `vartest`
- `$> cat vartest`

```
echo x = $x
echo y = $y
```

```
$> x=100
$> y=10
$> vartest
x =
y =
```

```
$> export y
$> vartest
x =
y = 10
```

(...) and {...} Constructs

- One or more commands inside paranthesis are executed in a subshell
  - `$> ( cd bin; ls; )`
    - List files in directory bin, which does not change cwd
  - `$> (prog1; prog2; prog3) 2>error.txt &`
    - Execute three programs in the background
    - Write errors from three programs to file error.txt
  - `$> x=50; (x=100); echo $x outputs 50`
- One or more commands inside curly braces are executed by the *current* shell
  - `$> { cd bin; ls; }`
    - List files in directory bin, which DOES change cwd
  - `$> x=50; { x=100; }; echo $x outputs 100`
- Input and output can be piped to and from these constructs, and I/O can be redirected.

Command Line Arguments

- If arguments are passed to a script, they have the values `$1,$2,$3,`etc.
- `$0` is the name of the script
- `$*` is a string of all of the arguments separated by spaces, excluding `$0`
- `$@` is an array of the arguments, excluding `$0`
- `$#` is the number of arguments

Output and Quoting

- `echo message` # print to stdout
- `echo –n "yes/no? "` # a prompt
  - Does not print newline after output
- Shell interprets $ and ‘ ‘ within double quotes
  - $ — variable substitution
  - ‘ — command substitution
  - `` echo "`date +%D`" `` # 04/30/05
- Shell does not interpret special characters within single quotes
  - `` echo '`date +%D`' `` # `date +%D`
- \ is used to escape characters (e.g. \”, \$)

Return Values

- Scripts can return an integer value
- Use `return N`
- The variable $? will contain the return value of the last command run
- Can be used to test conditions

```
$> pwd              $> pwdd
/home/user          pwdd: not found
$> echo $?          $> echo $?
0                   127
```

User-defined Variables

- **Examples:**
  - $>name=Ali #variable name is assigned a value Ali
  - $>echo $name #Ali will be displayed
  - $>echo Hello $name! , Welcome to $HOME #see output of this in your computer
- **Variable names:**
  - _FRUIT, TRUST_NO_1, _2_TIMES → Valid
  - 2_TIMES, _2*2, NO-1 → Invalid

User-defined Variables

- **Variable values:**
  - FRUIT=peach
  - FRUIT=2apples
  - FRUIT=apple+pear+kiwi
  - Be careful about spaces

    ```
    $> FRUIT=apple orange plum
    bash: orange: command not found.
    ```
  - Use quotes

    ```
    $> FRUIT="apple orange plum"
    ```

Reading User Input

- The general format: `read <variables>`
- When `read` is executed, the shell
  - reads a line from standard input
  - assigns the first word read to the first variable listed in <variables>
  - assigns the second word read to the second variable
  - and so on.
- If there are more words on the line than there are variables listed, the excess words get assigned to the last variable.
  - `read x y` reads a line from input, storing the first word read in the variable `x`, and the remainder of the line in the variable `y`.
- **Example**
  - **$> cat read.sh**

```
echo –n "Please enter your name and surname: "
read name1 name2
          echo "Welcome to CE Dept, KTU, $name1  $name2"
```

Command and Arithmetic Substitution

- Replacing with stdout from command
  - **var=`command`** (where ‘ ‘ is back quote)
  - **var=$(command)**
- Replacing with value of expression
  - **var=$((expresssion))**
- **Examples:**
  - $> echo ‘date‘ # It will display the output of date command
  - $> echo there are ‘who | wc –l‘ users working on the system # see output of this
  - c=$((2+3*4)) # "echo $c" displays 14

Integer Arithmetic

- Bash support evaluating arithmetic expressions without arithmetic substitution.
- The syntax is similar to $((...)) without the dollar sign.
  - $> x=10
  - $> ((x = x * 12))
  - $> echo $x   # gives 120
- Arithmetic expressions can be used in if, while and until commands.
- The comparison operators set the exit status to
  - a nonzero value if the result of the comparison is false
  - a zero value if the result is true
  - ((i == 100)) returns an exit status of zero (true) if i equals 100 and one (false) otherwise.
  - if ((i == 100)) ... has the same effect as if [“$i” –eq 100] ...

expr command

- `expr` command provides other forms for performing computations on user-defined variables
  - expr val1 op val2 (separated by spaces)
    - *where op is operator*
  - expr $val1 op $val2
  - val3=`expr $val1 op $val2`

Examples

```
$>expr 5 + 7                # gives 12
$>expr 6 – 3 – 2            # gives 1
$>expr 3 + 4 \* 5           # gives  23
$>expr 24 / 3               # gives  8
$>sum=`expr 5 + 6`
$>echo $sum                 # gives 11
$>a=12
$>b=90
$>echo sum is $a + $b       # sum is 12 + 90
$>echo sum is `expr $a + $b` # sum is 102
```

Predefined Variables

- There are some variables which are set internally by the shell and which are available to the user:
- $1 - $9 : Positional parameters
- $0 : Command name
- $# : Number of positional arguments
- $? : Exit status of the last command executed is given as a decimal string (0,1,2..).
- $$: Process number of this shell, useful for constructing unique filenames.

Predefined Variables

- $! : Process id of the last command run in the background (It holds PID of last background process).
- $- : Current options supplied to this invocation of the shell.
- $* : A string containing all the

  arguments to the shell, starting at $1.
- $@ : Same as above, except when quoted.

Predefined Variables

- **Notes:**
  - $* and $@ when unquoted are identical and expand into the arguments.
  - "$*" is a single word, comprising all the arguments to the shell, joined together with spaces. For example '1 2' 3 becomes "1 2 3".
  - "$@" is identical to the arguments received by the shell, the resulting list of words completely match what was given to the shell. For example '1 2' 3 becomes "1 2" "3".

Passing Arguments

- **Like standard UNIX commands, shell scripts can take arguments from the command line.**
- Arguments are passed from the command line into a shell program using the positional parameters $1 through to $9.
- The positional parameter $0 refers to the command name or name of the executable file containing the shell script.
- All the positional parameters can be referred to using the special parameter $*.

Examples

- $ cat pass_arg
- *#Script to accept 5 numbers and display their sum.*

```
echo the parameters passed are : $1, $2, $3, $4, $5
echo the name of the script is : $0
echo the number of parameters passed are : $#
sum=`expr $1 + $2 + $3 + $4 + $5`
echo The sum is : $sum
```

shift Command?

- If more than 9 parameters are passed to a script, there are two alternatives to access parameters:
  - The notation `${n}`
  - `shift` command
- The shift command shifts the parameters one position to the left. On the execution of shift command the first parameter is overwritten by the second, the second by third and so on.

Examples

- Write a script, which will accept different numbers and finds their sum. The number of parameters can vary.
  - $ cat sum_arg

```
 sum=0
 while [  $# -gt 0  ]
 do
    sum=`expr  $sum  +  $1`
    shift
 done
echo sum is $sum
```

Examples

-
  ```
  #!/bin/bash
      echo "arg1=$1 arg2=$2 arg3=$3"
      shift
      echo "arg1=$1 arg2=$2 arg3=$3"
      shift
      echo "arg1=$1 arg2=$2 arg3=$3"
      shift
      echo "arg1=$1 arg2=$2 arg3=$3"
  ```

The Null Command

- The shell has a built-in null command
  - The format is simply
    - :
  - The purpose is to do nothing
- It is generally used to satify the requirement that a command appear, particularly in if commands.

```
if grep "^$system" ~/mail/systems > /dev/null
then
    :
else
    echo "$system is not a valid system"
    exit 1
fi
```

  - The shell requires that you write a command after the then.
  - If the system is valid, nothing is done

The && and | | Operators

- The shell has two special constructs that enable you to execute a command based on whether the preceding command succeeds or fails.
- The operator && executes the command(s) following it, if and only if the preceding command was successfully compiled.
  - command1 && command2
  - command2 gets executed only if command1 returns an exit status of zero.
- Example
  - [-z $EDITOR] && EDITOR=/bin/ed

The && and || Operators

- The operator || executes the command(s) following it, if the preceding command failed.
  - command1 || command2
  - command2 gets executed only if command1 returns an exit status of nonzero.
- Examples
  - [-z $PATH] || echo $PATH
  - grep "$name" phonebook || echo \
    “Not found $name“
  - who | grep "^$name " > /dev/null || echo \
    "$name's not logged on“

  (Recall that when \ is used at the end of the line, it signals line continuation to the shell.)

The && and || Operators

- The && and || can also be combined on the same command line:

  -
    ```
    who | grep "^$name " > /dev/null && \
    echo "$name is logged on" || echo "$name’s \
    not logged on"
    ```
  - The first echo gets executed if the grep succeeds; the second if it fails.
- These operators can be represented with if commands

```
 if grep "$name" phonebook
then
    :
else
    echo "Couldn't find $name“
fi
```

Conditional Statements

- Every Unix command returns a value on exit, which the shell can interrogate. This value is held in the read-only shell variable $?.
- A value of `0` (zero) signifies success; anything other than `0` (zero) signifies failure.

Conditions

- If using integers: `(( condition ))`
- If using strings: `[[ condition ]]`
- Their exit status is zero or nonzero, depending on the condition.
- Examples:
  - `(( a  == 10 ))`
  - `(( b >= 3 ))`
  - `[[ $1 = -n ]]`
  - `[[ ($v != fun) && ( $v  != games) ]]`
  - `(( Z > 23 )) && echo Yes`

Conditions (cont.)

- Special conditions for file existence, file permissions, ownership, file type, etc.
- `[[ -e  $file ]]` –File exists?
- `[[ -f  $file ]]` –Regular file?
- `[[ -d  $file ]]` –Directory?
- `[[ -L  $file ]]` –Symbolic link?
- `[[ -r  $file ]]` –File has read permission?
- `[[ -w  $file ]]` –File has write permission?
- `[[ -x  $file ]]` –File has execute permission?
- `[[ -p  $file ]]` –File is a pipe?

The if statement

- The **`if`** statement uses the exit status of the given command and conditionally executes the statements following.
- **The general syntax is:**

```
if test
then
   commands     (if condition is true)
else
   commands     (if condition is false)
fi
```

Nested if statement

- **Nested if statement:**

```
if (-----)
then ...
else if ...
      ...
  fi
fi
```

- The **`elif`** statement can be used as shorthand for an **`else if`** statement.

if statements

- Syntax:

```
if condition
then
   commands
elif condition
then
   commands
else
    commands
fi
```

optional

if statement

- Example

```
if [[ -r  $fname ]]
then
  echo “$fname is readable”
elif [[ -w $fname && -x $fname ]]
then
  echo “$fname is writeable and
  executable”
fi
```

test Command

- The Unix system provides **test** command, which investigates the exit status of the previous command, and translates the result in the form of success or failure, i.e either a 0 or 1.
- The **test** command does not produce any output, but its exit status can be passed to the **if** statement to check whether the test failed or succeeded.

test Command (Cont.)

- **How to know exit status of any command?**
- All commands return the exit status to a pre-defined Shell Variable **‘?’,** which can be displayed using the **echo** command. **e.g**
- echo $?
- If output of this is 0 (Zero) it means the previous command was successful and if output is 1 (One) it means previous command failed.

test Command (cont.)

- The **test** command has specific operators to operate on files, numeric values and strings, which are explained below:
- **Operators on Numeric Variables used with test command:**

-eq	: equal to
-ne	: not equals to
-gt	: grater than
-lt	: less than
-ge	: greater than or equal to
-le	: less than or equal to

test Command (cont.)

- **Operators on String Variables used with test command:**

  = : equality of strings

  != : not equal

  -z : zero length string (i.e. string containing zero character
  i.e. null string).

  -n : String length is non zero.

Examples

```
$> a=12; b=23
$> test $a –eq $b
$> echo $?                   # gives 1
$> name=”Ahmet”
$> test –z $name             # returns 1
$> test –n $name             # returns 0
$> test –z “$address”        # returns 0
$> test $name = “Ali”        # returns 1
```

test Command (cont.)

- **Operators on files used with test command:**

  -f : the file exists.

  -s : the file exists and the file size is non zero.

  -d : directory exists.

  -r : file exists and has read permission.

  -w : file exists and has write permission.

  -x : file exists and has execute permission.

Examples

- $> test –f “mydoc.doc”

  # checks for the file mydoc.doc , if exists, returns 0 else 1.

- $> test –r “mydoc.doc”

  # checks for read permission for mydoc.doc

- $> test –d “$HOME”

  # checks for the existence of the users home directory.

test Command (cont.)

- **Logical Operators used with test command:**
- Combining more than one condition is done through the logical AND, OR and NOT operators.

-a	: logical AND
-o	: logical OR
!	: logical NOT

- $> test –r “mydoc.doc” –a –w “mydoc.doc”

  # checks both the read and write permission for the file mydoc.doc and returns either 0 or 1 depending on result.

test Command (cont.)

- To carry out a conditional action:

```
if who | grep -s ahmet > /dev/null
then
     echo ahmet is logged in CE server
else
     echo ahmet not available in CE server
fi
```

- This lists who is currently logged on to the system and pipes the output through grep to search for the username john

case statements

- Syntax:

```
case expression in
 pattern1)
  commands ;;
 pattern2)
  commands ;;
 …
 *)
  commands ;;
esac
```

case example 1

```
case $1 in
  -a)
     cmds related to option a ;;
  -b)
     cmds related to option b ;;
  *)
     all other options ;;
esac
```

case example 2

```
clear
echo "1. Date and time"
echo
echo "2. Directory listing"
echo
echo "3. Users information "
echo
echo "4. Current Directory"
echo
echo –n "Enter choice (1,2,3 or 4):"
```

case example 2

```
read choice
case $choice in
  1)   date;;
  2)   ls -l;;
  3)   who ;;
  4)   pwd ;;
  *)   echo wrong choice;;
esac
```

for loops

- Syntax:

```
for var [in list ]
do
   commands
done
```

- Commands on a single line are separated by semicolon (;).
- If *list* is omitted, $@ is assumed
- Otherwise `${list [*]}`
  - `where list is an array variable.`

for example 1

```
for colors in Red Blue Green
  Yellow Orange Black Gray White
  do
    echo $colors
  done
echo
```

for example 2

```
# See if a number of people are logged in
for i in $*
 do
   if who | grep -s $i > /dev/null
   then
     echo $i is logged in
   else
     echo $i not available
   fi
 done
```

while loops

- Syntax:

```
while command-list1
do
  command-list2
done
```

- if the exit status of the last command in *command-list1* is 0 (zero), the commands in *command-list2* are executed.
- The keywords break, continue, and return have the same meaning as in C/C++.
  - break [num] OR continue [num]   # num is number of loops

until loops

- Syntax:

```
Until command-list1
do
 command-list2
done
```

- The loop is executed as long as the exit status of *command-list1* is non-zero.
- The exit status of a while/until command is the exit status of the last command executed in *command-list2*. If no such command list is executed, a while/until has an exit status of 0.

Example 1

```
#!/bin/bash
while echo "Please enter command"
  read response
  do
    case "$response" in
      'done') break        ;; # no more commands
          "") continue      ;; # null command
           *) eval $response ;; # do the command
     esac
  done
```

Example 2

```
# To show use of case statement
echo What kind of tree bears acorns \?
read responce
case $responce in
  [Oo][Aa][Kk]) echo $responce is correct ;;
  *) echo Sorry, response  is wrong
esac
```

Example 3

```
# To show use of while statement
clear
echo What is the Capital of Turkey \?
read answer
while test $answer != Ankara
  do
    echo No, wrong please try again.
    read answer
  done
echo This is correct.
```

Example 4

```
# Accept the login name from the user
clear
echo "Please Enter the user login name: \n"
read login_name
until who | grep $login_name
  do
    sleep 5
    echo "Wrong name! Please try again: \n"
    read login_name
  done
echo The user $login_name has logged in
```

eval command

- Format: *eval command-line*
  - *command-line* is a normal command line
  - The shell scans the command line twice before executing it

```
$> eval echo hello  # displays hello
```

- Usage?

```
$> pipe="|"
$> ls $pipe wc –l
|: No such file or directory
wc: No such file or directory
-l: No such file or directory
```

- The shell takes care of pipes and I/O redirection before variable substitution, so it never recognizes the pipe symbol inside pipe.

```
$> eval ls $pipe wc –l  # displays 16
```

eval command

- The first time the shell scans the command line, it makes substitutions. Then eval causes it to rescan the line.
- If the variables contain command terminator (;, |, &), I/O redirection (<, >), and quote characters, eval can be useful.

```
$> cat last
eval echo \$$#
```

```
$> last one two three four
four
```

```
$> last *            # gets the last file
zoo_report
```

select loop

- Syntax:

```
select name in word1 ... wordN
do
  list
done
```

- *name* is the name of a variable.
- *word1* to *wordN* are sequences of characters separated by spaces (words). The set of commands to execute after the user has made a selection is specified by *list*.

select loop (cont.)

The execution process for select is as follows:

1. Each item is displayed along with a number.

2. A prompt, usually #?, is displayed.

3. $REPLY is set to a value entered by a user.

4. If $REPLY contains a valid number, *name* is set to the item selected. Otherwise, the items are displayed again.

5. When a valid selection is made, *list* executes.

6. If *list* does not exit from the select loop (i.e with break), the process starts over at step 1.

select loop (cont.)

- If the user enters more than one valid value, $REPLY contains all the user's choices. In this case, *name* is not set.
- You can change the prompt (#?) displayed by the select loop by altering the variable PS3. Otherwise, $PS3 is used as the prompt to display. For example, the commands

```
$> PS3="Please make a selection =>
"

$> export PS3
```

Example

```
select COMP in comp1 comp2 all none
do
  case $COMP in
    comp1|comp2) CompConf $COMP ;;
    all) CompConf comp1
         CompConf comp2 ;;
    none) break ;;
    *) echo “ERROR: Invalid selection,
   $REPLY." ;;
  esac
done
```

Example (cont.)

The menu presented by select to the user

```
1) comp1
2) comp2
3) all
4) none
#?
```

Using basename

- basename takes an absolute or relative path and returns the file or directory name.
- Its basic syntax

  basename *file*
- For example,

  $> basename /usr/bin/sh

  sh
- Using basename, you can define a variable USAGE in a script as follows:

  USAGE="Usage: ´basename $0´ [file|directory]"
- Then you can use it like echo "$USAGE"

Using `printf`

- It is similar to echo
- The only difference is that echo prints the newline automatically.

  $> echo "Is that a mango?" is identical to $> printf "Is that a mango?\n"
- Unlike echo, printf can perform complicated formatting using format specifications.

Using `printf` (cont.)

- The basic syntax for formatting:

  printf “*format*” *arg1 arg2 …*
- *format* is a string that describes how the remaining arguments are to be displayed.
- *arg1, arg2* are strings that correspond to the formatting characters specified in *format*.
- It is identical to the C language printf function.
- The formatting characters have the form:

  %[-,+,#]*m.nx*
- % starts the formatting character and *x* identifies the formatting character type.

Using `printf` (cont.)

- Depending on the value of *x*, the integers *m* and *n* are interpreted differently.
- *m* is the minimum length of a field, and *n* is the maximum length of a field.
- If you specify a real number format, *n* is treated as the precision that should be used.
- – left justifies the value being printed (by default, all fields are right justified).
- + causes printf to precede integers with a + or – sign (by default, only negative integers are printed with a sign).
- # causes printf to precede octal integers with 0 and hexadecimal integers with 0x or 0X for %#x or %#X, respectively.

Using `printf` (cont.)

- *x* can have the following possible values:

Letter	**Description**
d	Decimal numbers (integers)
u	Unsigned integers
o	Octal integers
x	Hexadecimal integers, using a-f
X	Hexadecimal integers, using A-F
c	Single characters
s	Literal strings
b	Strings containing backslash escape characters
%	Exponential floating-point number
e	Percent signs

Example 1

-
  ```
  $> printf "The octal value for %d is %o\n" 20 20
  The octal value for 20 is 24
  ```

-
  ```
  $> printf "The hexadecimal value for %d is %x\n" 30 30
  The hexadecimal value for 30 is 1e
  ```

-
  ```
  $> printf "The unsigned value for %d is %u\n" –1000 –1000
  The unsigned value for -1000 is 4294966296
  ```

-
  ```
  $> printf "This string contains a backslash escape: %s\n" "test\nstring“
  This string contains a backslash escape: test\nstring
  ```

-
  ```
  $> printf "This string contains an interpreted escape: %b\n" "test\nstring“
  This string contains an interpreted escape: test string
  ```

Example 2

-
  ```
  $> printf "%+d %+d %+d\n" 10 -10 20
  +10 -10 +20
  ```

-
  ```
  $> printf "%#o %#x\n" 100 200
  0144 0xc8
  ```

-
  ```
  $> printf "%20s%20s\n" string1 string2
               string1             string2
  ```

-
  ```
  $> printf "%-20s%-20s\n" string1 string2
  string1             string2
  ```

-
  ```
  $> printf "%5d%5d%5d\n" -1 -10 -100
     -1   -10   –100
  ```

-
  ```
  $> printf "%-5d%-5d%-5d\n" 1 10 100
  ```

-
  ```
  1    10   100
  ```

Example 3

-
  ```
  $> printf "%.5d %.4X\n" 10 27
  00010 001B
  ```

-
  ```
  $> printf "%.5s\n" abcdefg
  abcde
  ```

-
  ```
  $> printf ":%#10.5x:%5.4x:%5.4d\n" 1 10 100
  :   0x00001: 000a: 0100
  ```

-
  ```
  $> printf ":%9.5s:\n" abcdefg
  :    abcde:
  ```

-
  ```
  $> printf ":%-9.5s:\n" abcdefg
  :abcde    :
  ```

Output formatting example

```
#!/bin/bash
printf "%32s %s\n" "File Name" "File Type"
for i in *;
do
   printf "%32s " "$i"
   if [ -d "$i" ]; then echo "directory"
   elif [ -h "$i" ]; then echo "symbolic link"
   elif [ -f "$i" ]; then echo "file"
   else echo "unknown"
   fi;
done
```

Formatted output

```
 File Name   File Type
       RCS   directory
       dev   directory
     humor   directory
    images   directory
index.html   file
   install   directory
      java   directory
```

tee command

- Used to redirect output to a file, the screen or a pipe.
- Examples

```
$> date | tee now
$> ls | tee list | wc
$> ps -ael | tee processes | grep "$UID"
```

Reading Files

- The most common use of redirection is for reading files one line at a time.

-
  ```
  while read LINE          #using read command
  do
        : # manipulate file here
  done < file
  ```

-
  ```
  while LINE=`line`        #using line command
  do
        : # manipulate file here
  done < file
  ```

Reading Files (Cont.)

-
  ```
  #!/bin/bash
  if [ -f "$1" ] ; then
      i=0
      while read LINE
      do
          i=`echo "$i + 1" | bc`
      done < "$1"
      echo $i
  fi
  ```
- This script tries to count the number of lines in the file specified to it as an argument.

Other ways to read files (1)

- Using cat, *pipe* and read

```
cat file | while read LINE
do
	: # manipulate file here
done
```

- Using cat, *pipe* and line

```
cat file | while LINE=`line`
do
	: # manipulate file here
done
```

Other ways to read files (2)

- Using exec and read

```
exec 3<&0; exec 0< file
while read LINE
do
        : # manipulate file here
done
exec 0<&3
```

- Using exec and line

```
exec 3<&0; exec 0< file
while LINE=`line`
do
        : # manipulate file here
done
exec 0<&3
```