

# Sistemas Operativos

## Shell Scripts

Ricardo Ruiz Rodríguez

Instituto de Computación  
Universidad Tecnológica de la Mixteca  
Primavera 2014

# Índice I

## 1 Flow control (selection)

- The if statement
- The case statement

## 2 Flow control (loops)

- The while loop
- The for and select loops
- Loop control

## 3 Parameters

- Introduction
- Special variables
- Options, arguments and parsing

# Introduction I

- The order in which commands execute in a shell script is called *the flow* of the script.
- In the scripts that you have looked at so far, the flow is always the same because the same set of commands executes every time.
- In most scripts, you need to change the commands that execute depending on some condition provided by the user or detected by the script itself.
- When you change the commands that execute based on a condition, you change the *flow* of the script.
- Two powerful flow control mechanics are available in the shell:
  - ① The if statement.
  - ② The case statement.

## Introduction II

- The **if** statement is normally used for the conditional execution of commands.
- The **case** statement enables any number of command sequences to be executed depending on which one of several patterns matches a variable first.

# The if statement I

- The if statement performs actions depending on whether a given condition is true or false.
- Because the return code of a command indicates true (return code is zero) or false (return code is nonzero), one of the most common uses of the if statement is in error checking.
- The basic **if** statement syntax follows:

```
if list1 then
    list2
elif list3 then
    list4
else
    list5
fi
```

## The if statement II

- Both the **elif** and the **else** statements are optional.
- If you have an **elif** statement, you don't need an **else** statement and vice versa.
- An **if** statement can be written with any number of **elif** statements.
- The *flow control* for the general **if** statement follows:
  - ① *list1* is evaluated.
  - ② If the exit code of *list1* is 0, indicating a true condition, *list2* is evaluated and the **if** statement exits.
  - ③ Otherwise, *list3* is executed and its exit code is checked.
  - ④ If *list3* returns 0, *list4* executes and the **if** statement exits.
  - ⑤ If *list3* does not return 0, *list5* executes.
- Because the shell considers an **if** statement to be a list, you can write it all in one line as follows:

## The if statement III

```
if list1 ; then list2 ; elif list3 ; then list4 ;  
    else list5 ; fi ;
```

- Usually this form is used only for short **if** statements.

# The if statement examples I

- See example *ex-10-01\_if.sh*
- See example *ex-10-02\_if.sh*
- Four common errors can occur when using the **if** statement:
  - ❶ Omitting the semicolon (;) before the **then** statement in the single line form.
  - ❷ Using *else if* or *elsif* instead of **elif**.
  - ❸ Omitting the **then** statement when an **elif** statement is used.
  - ❹ Writing *if* instead of **fi** at the end of an **if** statement.



# Using test |

- Most often, the list given to an **if** statement is one or more test commands, which are invoked by calling the **test** command as follows:

**test** expression

Here *expression* is constructed using one of the *special options* to the **test** command.

- The **test** command returns either a 0 (true) or a 1 (false) after evaluating an expression.
- A *shorthand* for the **test** command is the **[** command:  
[ expression ]

Here *expression* is any valid expression that the **test** command understands.

## Using test II

- This shorthand form is the most common form of test that you can encounter.
- The types of expressions understood by **test** can be broken into three types:
  - 1 File tests.
  - 2 String comparisons.
  - 3 Numerical comparisons.
- When using the **[** shorthand for test, the space after the open bracket (**[**) and the space before the close bracket (**]**) are required.

# File tests I

- File test expressions test whether a file fits some particular criteria.
- The general syntax for a file test is:

**test** option file

or

[ option file ]

Here *option* is one of the **test** command and *file* is the name of a file or directory.

The file test options for the test command are the following:

- **-b file**: true if file exists and is a block special file.
- **-c file**: true if file exists and is a character special file.
- **-d file**: true if file exists and is a directory.

## File tests II

- **-e file**: true if file exists.
- **-f file**: true if file exists and is a regular file.
- **-g file**: true if file exists and has its SGID bit set.
- **-h file**: true if file exists and is a symbolic link.
- **-k file**: true if file exists and has its "sticky" bit set.
- **-p file**: true if file exists and is a named pipe.
- **-r file**: true if file exists and is readable.
- **-s file**: true if file exists and has a size greater than zero.
- **-u file**: true if file exists and has its SUID bit set.
- **-w file**: true if file exists and is writable.
- **-x file**: true if file exists and is executable.
- **-O file**: true if file exists and is owned by the effective user ID.

## File tests III

```
if [ -d /home/rruiz/bin ] ; then  
    PATH="$PATH:/home/rruiz/bin" ;  
fi
```

# String comparisons I

- The **test** command also supports simple string comparisons.
- There are two main forms:
  - ① Checking whether a string is empty.
  - ② Checking whether two strings are equal.
- *A string cannot be compared to an expression using the test command.* The case statement, covered later, has to be used instead.

The test options relating to string comparisons are the following:

- ① **-z string**: true if *string* has zero length.
- ② **-n string**: true if *string* has nonzero length.
- ③ **string1 = string2**: true if the *strings* are equal.
- ④ **string1 != string2**: true if the *strings* are not equal.

## String comparisons II

- The syntax of the first form is:

**test** option string

or

[ option string ]

Here *option* is either **-z** or **-n** and *string* is any valid shell string.

- See example *ex-10-03\_test.sh*.
- Notice that the variable `$FRUIT_BASKET` is quoted in this example. This is required in the event that the variable is unset (why?).
- The shell does not quote a null value and:  
$$[-z] \neq [-z ""]$$
- The **test** command enables you to determine whether two strings are equal.

## String comparisons III

- Two strings are considered equal if they contain exactly the same sequence of characters.
- The basic syntax for checking whether two strings are equal is:

**test** *string1* = *string2*

or

**[** *string1* = *string2* **]**

Here *string1* and *string2* are the two strings being compared.

- See example *ex-10-03\_test.sh*.



# Numerical comparisons I

- The **test** command enables you to compare two integers.
- The basic syntax is:

**test** int1 operator int2

or

**[** int1 operator int2 **]**

Here *int1* and *int2* can be any positive or negative integers and operator is one of the following operators:

- ❶ **int1 -eq int2**: true if *int1* **equals** *int2*.
- ❷ **int1 -ne int2**: true if *int1* is **not equal** to *int2*.
- ❸ **int1 -lt int2**: true if *int1* is **less than** *int2*.
- ❹ **int1 -le int2**: true if *int1* is **less than or equal** to *int2*.
- ❺ **int1 -gt int2**: true if *int1* is **greater than** *int2*.
- ❻ **int1 -ge int2**: true if *int1* is **greater than or equal** to *int2*.

## Numerical comparisons II

- If either `int1` or `int2` is a string, not an integer, it is treated as 0.
- Among the most common tasks in a shell script are executing a program and checking its return status.
- By using the numerical comparison operators, you can check the return or exit status of a command and perform different actions when a command is successful and when a command is unsuccessful.
- If you execute this command on the command line, you can see any error messages and intervene to fix the problem.
- In a shell script, the error message is ignored and the script continues to execute. For this reason, it is necessary to check whether a program exited successfully.

## Numerical comparisons III

- As you saw with the **test** command, an exit status of 0 is successful, whereas nonzero values indicate some type of failure.
- The exit status of the last command is stored in the variable `$?`, so you can check whether a command was successful as follows:

```
if [ $? -eq 0 ] ; then
    echo "Command was successful." ;
else
    echo "An error was encountered."
    exit
fi
```

# Compound expressions I

- So far you have seen individual expressions, but many times you need to combine expressions in order to satisfy a particular expression.
- When two or more expressions are combined, the result is called a **compound expression**.
- You can create compound expressions using the test command's built in operators, or you can use the conditional execution operators `&&` and `||`.
- Also you can create a compound expression that is the negation of another expression by using the `!` operator.
- Operators for creating compound expressions are the following:

## Compound expressions II

- ① **! *expr***: true if *expr* is false. The *expr* can be any valid test command.
- ② ***expr1* -a *expr2***: true if both *expr1* **and** *expr2* are true.
- ③ ***expr1* -o *expr2***: true if either *expr1* **or** *expr2* is true.
- The syntax for creating compound expressions using the built-in operators is:

**test** *expr1* operator *expr2*

or

**[** *expr1* operator *expr2* **]**

Here *expr1* and *expr2* are any valid test expression, and operator is either **-a** (a as in **and**) or **-o** (o as in **or**).

## Compound expressions III

- The syntax for creating compound expressions using the conditional operators is:

**test** *expr1* **operator** **test** *expr1*

or

[ *expr1* ] **operator** [ *expr2* ]

Here *expr1* and *expr2* are any valid test expression, and operator is either && (**and**) or || (**or**).

```
if [ -z "$DTHOME" ] && [ -d /usr/dt ] ; then
    DTHOME=/usr/dt ;
fi
```

```
if [ -z "$DTHOME" -a -d /usr/dt ] ; then
    DTHOME=/usr/dt ;
fi
```

## Compound expressions IV

- The final type of compound expression consists of negating an expression.
- Negation reverses the result of an expression. True expressions are treated as false expressions and vice versa.
- The basic syntax of the negation operator is:

**test ! expr**

or

**[ ! expr ]**

Here *expr* is any valid test expression.

- A simple example is the following command:

## Compound expressions V

```
if [ ! -d $HOME/bin ] ; then  
    mkdir $HOME/bin ;  
fi
```

```
test ! -d $HOME/bin && mkdir $HOME/bin
```



# Syntax I

- The **case** statement is the other major form of *flow control* available in the shell.
- The basic syntax is:

```
case word in
    pattern1)
        list1
        ;;
    pattern2)
        list2
        ;;
esac
```

- Here the string *word* is compared against every *pattern* until a match is found.

## Syntax II

- The *list* following the matching pattern executes. If no matches are found, the **case** statement exits without performing any action.
- There is no maximum number of patterns, but the minimum is one.
- When a *list* executes, the command **;;** indicates that program flow should jump to the end of the entire **case** statement. This is similar to **break** in the C programming language.
- Some programmers prefer to use the form:

```
case word in
    pattern1) list1 ;;
    pattern2) list2 ;;
esac
```

## Syntax III

- This form should be used only if the list of commands to be executed is short.
- See example *ex-10-04\_case.sh*

# Using patterns I

- In the example (first section), you used fixed strings as the pattern.
- If used in this fashion the case statement degenerates into an if statement. For example, the if statement:

```
if [ "$FRUIT" = apple ] ; then
    echo "Apple pie is quite tasty."
elif [ "$FRUIT" = banana ] ; then
    echo "I like banana nut bread."
elif [ "$FRUIT" = kiwi ] ; then
    echo "New Zealand is famous for kiwi."
fi
```

## Using patterns II

is more verbose, but the real power of the **case** statement does not lie in simplifying if statements.

- The power of **case** statement lies in the fact that it uses patterns to perform matching.
- A **pattern** is a string that consists of regular characters and special wildcard characters.
- The pattern determines whether a match is present.
- The patterns can use the same special characters as patterns for pathname expansion covered in "Substitution".
- along with the or operator |.
- Some default actions can be performed by giving the \* pattern, which matches anything.

## Using patterns III

```
case "$TERM" in
    *term)
        TERM=xterm ;;
    network|dialup|unknown|vt[0-9][0-9][0-9])
        TERM=vt100 ;;
esac
```

- See example *ex-10-04\_case.sh*

# Introduction I

- Loops enable you to execute a series of commands multiple times.
- Two main types of loops are:
  - ① The **while** loop.
  - ② The **for** loop.
- The **while** loop enables you to execute a set of commands repeatedly until some condition occurs.
- It is usually used when you need to manipulate the value of a variable repeatedly.
- The **for** loop enables you to execute a set of commands repeatedly for each item in a list.
- One of its most common uses is in performing the same set of commands for a large number of files.

## Introduction II

- In addition to these two types of loops, *ksh* and *bash* support an additional type of loop called the **select** loop.
- The **select** loop frequently presents a menu of choices to a shell scripts user.



# The while loop I

- The basic syntax of the **while** loop is:

```
while command
do
    list
done
```

Here *command* is a single command to execute, whereas *list* is a set of one or more commands to execute.

- Although *command* can be any valid UNIX command, it is usually a test expression.
- *list* is commonly referred to as the *body* of the **while** loop because it contains the heart of the loop.

## The while loop II

- The **do** and **done** keywords are not considered part of the body of the loop because the shell uses them only for determining where the **while** loop begins and ends.
- If both *command* and *list* are short, the **while** loop is written in a single line as follows:

```
while command ; do list ; done
```

- See example *ex-11-01\_while.sh*

# Nesting while loops I

- It is possible to use a **while** loop as part of the body of another **while** loop as follows:

```
while command1 ; # this is loop1, the outer loop
do
    list1
    while command2 ; # this is loop2, the inner loop
    do
        list2
    done
    list3
done
```

## Nesting while loops II

Here *command1* and *command2* are single commands to execute, whereas *list1*, *list2*, and *list3* are a set of one or more commands to execute. Both *list1* and *list3* are optional.

- Here you have two **while** loops, *loop1* and *loop2*. Usually *loop1* is referred to as the *main* loop or *outer* loop, and *loop2* is referred to as the *inner* loop.
- When describing the *inner* loop (*loop2*), many programmers say that it is *nested* one level deep.
- The term **nested** refers to the fact that *loop2* is located *in the body* of *loop1*.
- If you had a *loop3* located in the body of *loop2*, it would be *nested* two levels deep. The *level of nesting* is *relative* to the *outermost* loop.

## Nesting while loops III

- There are no restrictions on how deeply nested loops can be, but you should try to avoid nesting loops more deeply than four or five levels to avoid difficulties in finding and fixing problems in your script.
- See example `ex-11-02_while.sh` and `ex-11-02-2_while.sh`

# Validating user input I

- Say that you need to write a script that needs to ask the user for the name of a directory. You can use the following steps to get information from the users:
  - ① Ask the user a question.
  - ② Read the user's response.
  - ③ Check to see whether the user responded with the name of a directory.
- What should you do when the user gives you a response that is not a directory?
- One of the most common uses for the **while** loop is to check whether user input has been gathered correctly. Usually a strategy similar to the following is employed:
  - ① Set a variable's value to null.
  - ② Start a **while** loop that exits when the variable's value is not null.

## Validating user input II

- ③ In the **while** loop, ask the user a question and read in the users response.
  - ④ Validate the response.
  - ⑤ If the response is invalid the variable's value is set to null. This enables the **while** loop to repeat.
  - ⑥ If the response is valid, the variable's value is not changed. It continues to hold the user's response. Because the variable's value is not null, the while loop exits.
- See example *ex-11-03\_while.sh*.

# The until loop I

- The **while** loop is perfect for a situation where you need to execute a set of commands *while* some condition is true.
- Sometimes you need to execute a set of commands *until* a condition is true.
- A *variation* on the **while** loop *available only* in **ksh** and **bash**, the **until** loop provides this functionality.
- Its basic syntax is:

```
until command
do
    list
done
```



# The until loop II

Here *command* is a single command to execute, whereas *list* is a set of one or more commands to execute.

- Although *command* can be any valid UNIX command, it is usually a **test** expression.
- If both *command* and *list* are short, the **until** loop can be written on a single line as follows:

```
until command ; do list ; done
```

- See example *ex-11-03\_until.sh*.
- The **until** loop offers no advantages over the equivalent **while** loop.

# The for and select loops

- Unlike the **while** loop, which exits when a certain condition is false, both the **for** and **select** loops operate on *lists* of items.
- The **for** loop *repeats* a set of commands for every item in a list.
- The **select** loop enables the user to *select* an item from a list.

# The for loop I

- The basic syntax is:

```
for name in word1 word2 ... wordN
do
    list
done
```

Here *name* is the name of a variable and *word1* to *wordN* are sequences of characters separated by spaces (words).

- Each time the **for** loop executes, the value of the variable *name* is set to the next word in the list of words, *word1* to *wordN*. The first time, name is set to word1; the second time, it's set to word2; and so on.

# The for loop II

- This means that the number of times a **for** loop executes depends on the number of words that are specified.
- In each iteration of the **for** loop, the commands specified in *list* are executed.
- You can also write the entire loop on a single line as follows:  

```
for name in word1 word2 ... wordN ; do list ; done
```

If *list* and the number of *words* are short, the single line form is often chosen; otherwise, the multiple-line form is preferred.

- See *example ex-11-05\_for.sh* and compares with *ex-11-01\_while.sh*.

# Manipulating a set of files I

- Say that you need to copy a bunch of files from one directory to another and change the permissions on the copy.
- What would you do?

⋮

## Manipulating a set of files II

- You could do this by copying each file and changing the permissions manually.
- A better solution would be to determine the commands you need to execute in order to copy a file and change its permissions and then have the computer do this for every file you were interested in.
- In fact this is one of the most common uses of the **for** loop: iterating over a set of file names and performing some operations on those files.
- The procedure to do this follows:
  - ① Create a **for** loop with a variable named *file* or *FILE*. Other favored names include *i*, *j*, and *k*. Usually the name of the variable is singular.

## Manipulating a set of files III

- ② Create a list of files to manipulate. This is frequently accomplished using the filename substitution technique discussed in "Substitution".
- ③ Manipulate the files in the body of the loop.
- See *example ex-11-06\_for2.sh*
- Notice that you are using the name `FILE` for the variable. This is because each time you are dealing with a single file from a list of files.
- The rationale behind making the for loop's variable singular, such as `FILE` instead of `FILES`, is that you are dealing with only one item from a set of items each time the loop executes.

# The select loop I

- The **select** loop provides an easy way to create a numbered menu from which users can select options.
- It is useful when you need to ask the user to choose one or more items from a list of choices.
- This loop was introduced in *ksh* and has been adapted into *bash*. It is not available in *sh*.
- The basic syntax of the **select** loop is:

```
select name in word1 word2 ... wordN
do
    list2
done
```



## The select loop II

Here *name* is the name of a variable and *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 *list2*.

- The execution process for a **select** loop is as follows:
  - ① Each item in *list1* is displayed along with a number.
  - ② A prompt, usually **#?**, is displayed.
  - ③ When the user enters a value, **\$REPLY** is set to that value.
  - ④ If **\$REPLY** contains a number of a displayed item, the variable specified by *name* is set to the item in *list1* that was selected. Otherwise, the items in *list1* are displayed again.
  - ⑤ When a valid selection is made, *list2* executes.
  - ⑥ If *list2* does not exit from the **select** loop using one of the loop control mechanisms such as *break*, the process starts over at step 1.

## The select loop III

- You can change the prompt displayed by the **select** loop by altering the variable **PS3**.
- If **PS3** is not set, the default prompt, **#?**, is displayed. Otherwise the value of **PS3** is used as the prompt to display.
- See example *ex-11-07\_select.sh*

# Loop control

- So far we have looked at creating loops and working with loops to accomplish different tasks.
- Sometimes you need to stop a loop or skip iterations of the loop.
- The commands used to control loops are:
  - 1 break.
  - 2 continue.

# The break command I

- If you make a mistake in specifying the termination condition of a **while** loop, it can continue forever.
- For example, say you forgot to specify the \$ before the x in the test expression:

```
x=0
while [ x -lt 10 ]
do
    echo $x
    x= `echo "$x + 1" | bc`
done
```

- This loop would continue to display numbers forever.

## The break command II

- A loop that executes forever without terminating executes an infinite number of times. For this reason, such loops are called **infinite loops**.
- In most cases infinite looping is not desired and stems from programming errors, but in certain instances they can be useful. For example, say that you need to wait for a particular event, such as someone logging on to a system, to occur.
- You can use an infinite loop to check every few seconds whether the event has occurred.
- Because you don't know how many times you need to execute the loop, when the event occurs, you can exit the infinite loop using the break command.
- In *sh*, you can create infinite loops using the **while** loop.

## The break command III

- Because a **while** loop executes *list* while command is true, specifying command as either `:` or `/bin/true` causes the loop to execute forever.
- The basic syntax of the **infinite** while loop is:

```
while :  
do  
    list  
done
```

- In most infinite loops, the while loop usually exits from within *list* via the **break** command, which enables you to exit any loop immediately.
- See example *ex-11-08\_break.sh*

## The break command IV

- The **break** command also accepts as an argument an integer, greater or equal to 1, indicating the number of levels to break out of.
- This feature is useful when *nested* loops are being used.

# The break command V

```
for i in 1 2 3 4 5
do
    mkdir -p /mnt/backup/docs/ch0${i}
    if [ $? -eq 0 ] ; then
        for j in doc c h m pl sh
        do
            cp $HOME/docs/ch0${i}/*.${j}
            /mnt/backup/docs/ch0${i}
            if [ $? -ne 0 ] ; then break 2 ; fi
        done
    else
        echo "Could not make backup directory."
    fi
done
```



# The continue command

- The **continue** command is similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop.
- This command is useful when an error has occurred but you want to try to execute the next iteration of the loop.
- The loop in the example *ex-11-10\_continue2.sh* does not exit if one of the input files is bad.
- See example *ex-11-10\_continue3.sh* also.

# Introduction I

- The general format for the invocation of programs in UNIX is:  
**command** options files

Here *command* is the command name, *options* is any option that you need to specify, and *files* is an optional list of files on which the command should operate.

- Because most UNIX users are familiar with this interface, you should adhere to this format in shell scripts.
- This means that scripts that can have options specified must be able to read and interpret them correctly.
- How should you address this?

## Introduction II

- You have two common methods for the handling options passed to a shell script:
  - ① Handle options manually using a **case** statement.
  - ② Handle options using the **getopts** command.
- For scripts that support only one or two options, the first method is easy to implement and works quite well.
- However, many scripts allow any combination of several options to be given. For such scripts, the **getopts** command is very useful because it affords the maximum flexibility in parsing options.

# Special variables I

- The shell defines several special variables that are relevant to option parsing.
- In addition to these, a few variables give the status of commands that the script executes.  
The special shell variables are:
- **\$0** The name of the command being executed. For shell scripts, this is the path with which it was invoked.
- **\$n** These variables correspond to the arguments with which a script was invoked. Here *n* is a positive decimal number corresponding to the position of an argument (the first argument is \$1, the second argument is \$2, and so on).
- **\$#** The number of arguments supplied to a script.

## Special variables II

- **\$\*** All the arguments are double quoted. If a script receives two arguments, **\$\*** is equivalent to **\$1 \$2**.
- **\$@** All the arguments are individually double quoted. If a script receives two arguments, **\$@** is equivalent to **\$1 \$2**.
- **\$?** The exit status of the last command executed.
- **\$\$** The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
- **#!** The process number of the last background command.

# Using \$0 I

- The \$0 variable is commonly used to determine the behavior of scripts that can be invoked with more than one name.
- Consider the following script:

```
#!/bin/sh
case $0 in
    *listtar) TARGETS="-tvf $1" ;;
    *maketar) TARGETS="-cvf $1.tar $1" ;;
esac
tar $TARGETS
```

## Using \$0 II

- You can use this script to list the contents of a tar file (*t* as in *tape* and *ar* as in *archive*, a common format for distributing files in UNIX) or to create a tar file based on the name with which the script is invoked.
- The tar file to read or create is specified as the first argument, \$1.
- I called this script *mytar* and made two symbolic links to it called *listtar* and *maketar* as follows:

```
$ ln -s mytar listtar
```

```
$ ln -s mytar maketar
```

- If the script is invoked with the name *maketar* and is given a directory or filename, a tar file is created.

## Using \$0 III

- If you had a directory called `fruits` with the following contents:

```
$ ls fruits  
apple banana mango peach pear
```

You can invoke the script as *maketar* to obtain a tar file called *fruit.tar* containing this directory, by issuing the following command:

```
$ ./maketar fruits
```

- If you want to list the contents of this tar file, you can invoke the script as follows:

```
$ ./listtar fruits.tar
```



## Using \$0 IV

- the output that you encounter depends on the version of tar that is installed on your machine. Some versions include more detail in the output than others.
- Another common use for \$0 is in the usage statement for a script.
- Usage statement is a short message informing the user how to invoke the script properly.
- All scripts used by more than one user should include such a message. In general, the usage statement is something like the following:

```
echo "Usage: $0 [options] [files]"
```

## Using \$0 V

- If you consider the *mytar* script given previously, a usage statement would be a helpful addition, in case the script was called with some name other than the two names it knows about.
- To implement this, change the case statement as follows:

```
case $0 in
    *listtar) TARGETS="-tvf $1" ;;
    *maketar) TARGETS="-cvf $1.tar $1" ;;
    *) echo "Usage: $0 [file|directory]"
       exit 0
       ;;
esac
```

## Using \$0 VI

- Thus, if the script is invoked as just *mytar*, you see following message:

Usage: mytar [file|directory]

- Although this message describes the usage of the script correctly, it does not inform us that the script's name was given incorrectly.
- There are two possible methods for rectifying this:
  - ① Hard coding the valid names in the "usage statement".
  - ② Changing the script to use its *arguments* to decide in which mode it should run.
- To demonstrate the use of options, the next section uses the latter method.

# Options and arguments

- Options are given on the command line to change the behavior of a script or program.
- Often you will see or hear options called *arguments*. The difference between the two is subtle.
- A command's **arguments** are all of the separate strings or words that appear on the command line after the command name, whereas **options** are only those arguments that *change* the behavior of the command.

# Dealing with arguments I

- To illustrate the use of options, change the mytar script to use its first argument, \$1, as the *mode* argument and \$2 as the tar file to read or create.
- To implement this, change the case statement as follows:

```
USAGE="Usage: $0 [-c|-t] [file|directory]"
```

```
case "$1" in
    -t) TARGETS="-tvf $2" ;;
    -c) TARGETS="-cvf $2.tar $2" ;;
    *) echo "$USAGE"
        exit 0
        ;;
esac
```

## Dealing with arguments II

- The three major changes are:
  - ① All references to \$1 have been changed to \$2 because the second argument is now the filename.
  - ② listtar has been replaced by -t.
  - ③ maketar has been replaced by -c.
- Now running mytar produces the correct output:  
Usage: ./mytar [-c|-t] [file|directory]
- To create a tar file of the directory fruits with this version, use the command:  

```
$ ./mytar -c fruits
```
- To list the contents of the resulting tar file, fruits.tar, use the command:  

```
$ ./mytar -t fruits
```

# Using basename I

- Currently, the message displays the entire path with which the shell script was invoked, but what is really required is the name of the shell script.
- You can correct this by using the **basename** command.
- The basename command takes an absolute or relative path and returns the file or directory name. Its basic syntax is:

**basename** file

- For example:

```
$ basename /usr/bin/sh
```

prints the following:

```
sh
```

- Using basename, you can change the variable \$USAGE in the mytar script as follows:

## Using basename II

```
USAGE="Usage: `basename $0` [-c|-t] [file|directory]"
```

This produces the following output:

```
Usage: mytar [-c|-t] [file|directory]
```

- You could also have used the `basename` command in the first version of the `mytar` script to avoid using the `*` wildcard character in the case statement as follows:

```
#!/bin/sh
case `basename $0` in
    listtar) TARGETS="-tvf $1" ;;
    maketar) TARGETS="-cvf $1.tar $1" ;;
esac
tar $TARGETS
```



## Using basename III

- In this version, the `basename` command allows us to match the exact names with which scripts can be called. This simplifies the possible user interactions and is preferred for that reason.
- As an illustration of a potential problem with the original version, you can see that if the script is called:

`$ ./makelisttar`

the original version would use the first case statement, even though it was incorrect, but the new version would fall through and report an error.

## Additional handling I

- Now that the mytar script uses options to set the mode in which the script runs, there is another problem to solve.
- What should it do if the second argument, \$2, is not provided?
- You don't have to worry about what happens if the first argument, \$1, is not provided because the case statement deals with this situation via the default case, \*.
- The simplest method for checking the necessary number of arguments is to see whether the number of given arguments, \$# , matches the number of required arguments.
- Add this check to the script:

## Additional handling II

```
#!/bin/sh
USAGE="Usage: `basename $0` [-c|-t] [file|directory]"
if [ $# -lt 2 ] ; then
    echo "$USAGE"
    exit 1
fi
case "$1" in
    -t) TARGETS="-tvf $2" ;;
    -c) TARGETS="-cvf $2.tar $2" ;;
    *) echo "$USAGE"
        exit 0
        ;;
esac
tar $TARGETS
```

## Additional handling III

- This mytar script is mostly finished, but you can still make a few improvements.
- For example, it only deals with the first file that is given as an argument, and it does not check to see whether the file argument is really a file.
- You can add the processing of all file arguments by using the special shell variable `$@`. Start with the `-t` (list contents) option.
- The case statement now becomes:

## Additional handling IV

```
case "$1" in
  -t) TARGETS="-tvf"
      for i in "$@" ; do
        if [ -f "$i" ] ; then tar $TARGETS "$i" ; fi ;
      done
      ;;
  -c) TARGETS="-cvf $2.tar $2" ;
      tar $TARGETS
      ;;
  *) echo "$USAGE" ;
      exit 0
      ;;
esac
```

## Additional handling V

- The main change is that the `-t` case now includes a for loop that cycles through the arguments and checks to see whether each one is a file. If an argument is a file, `tar` is invoked on that file.
- When examining the arguments passed to a script, two special variables are available for inspection, `$*` and `$@`.
- The main difference between these two is how they expand arguments.
- When `$*` is used, it simply expands each argument without preserving quoting. This can sometimes cause a problem. If your script is given a filename containing spaces as an argument:

```
mytar -t "my tar file.tar"
```

## Additional handling VI

- Using `$*` would mean that the `for` loop would call `tar` three times for files named *my*, *tar*, and *file.tar*, instead of once for the file you requested: *my tar file.tar*.
- By using `$@`, you avoid this problem because it expands each argument as it was quoted on the command line.
- You should deal with a few more minor issues.
- Looking closely, you see that all the arguments given to the script, including the first argument, `$1`, are considered as files.
- Because you are using the first argument as the flag to indicate the mode in which the script runs, you should not consider it.
- Not only does this reduce the number of times the `for` loop runs, but it also prevents the script from accidentally trying to run `tar` on a file with the name `-t`.

## Additional handling VII

- To remove the first argument from the list of arguments, use the shift command. A similar change to the make mode of the script is also required.
- Another issue is what the script should do when an operation fails.
- In the case of the listing operation, if the tar cannot list the contents of a file, skipping the file and printing an error would be a reasonable operation.
- Because the shell sets the variable `$?` to the exit status of the most recent command, you can use that to determine whether a tar operation failed.
- See example *mytar2*.



# Option parsing I

- You have two common ways to handle the parsing of options passed to a shell script.
- In the first method, you can manually deal with the options using a case statement. This method was used in the *mytar* script.
- The second method, is to use the **getopts** command.
- The syntax of the **getopts** command is:  
**getopts** option-string variable  
Here *option-string* is a string consisting of all the single character options **getopts** should consider, and *variable* is the name of the variable that the option should be set to.
- Usually the variable used is named **OPTION**.

## Option parsing II

The process by which getopt parses the options given on the command line is:

- ① The getopt option examines all the command line arguments, looking for arguments starting with the - character.
- ② When an argument starting with the - character is found, it compares the characters following the - to the characters given in the *option-string*.
- ③ If a match is found, the specified variable is set to the option: otherwise, variable is set to the ? character.
- ④ Steps 1 through 3 are repeated until all the options have been considered.
- ⑤ When parsing has finished, getopt returns a nonzero exit code. This allows it to be easily used in loops.

## Option parsing III

- ⑥ Also, when `getopts` has finished, it sets the variable `OPTIND` to the index of the last argument.
- ⑦ Another feature of **`getopts`** is the capability to indicate options requiring an additional parameter.
- ⑧ You can accomplish this by following the option with a `:` character in the *option-string*. In this case, after an option is parsed, the additional parameter is set to the value of the variable named `OPTARG`.

# Using getopt

- To get a feeling for how getopt works and how to deal with options, write a script that simplifies the task of uuencoding a file.
- uuencode it is a program that was originally used to encode binary files (executable files) into ASCII text so that they could be emailed or transferred via FTP.
- First, examine the interface of this script, which makes it easier to understand the implementation.
- The script should be able to accept the following options:
  - **-f** to indicate the input filename.
  - **-o** to indicate the output filename.
  - **-v** to indicate the script should be verbose.

## Using getopt II

- The **getopts** command to implement these requirements is:  
**getopts f:o:v OPTION**
- This indicates that all the options except -v require an additional parameter.
- The variables you require in order to support this are:
  - VERBOSE, which stores the value of the verbose flag. By default this is false.
  - INFILE, which stores the name of the input file.
  - OUTFILE, which stores the name of the output filename. If this value is unset, **uudecode** uses the name supplied in the input file, and **uuencode** uses the name of the supplied input file and append to it the **.uu** extension.
- The loop to implement the preceding requirements is as follows:

## Using getopt III

```
VERBOSE=false
while getopt f:o:v OPTION ;
do
    case "$OPTION" in
        f) INFILE="$OPTARG" ;;
        o) OUTFILE="$OPTARG" ;;
        v) VERBOSE=true ;;
        \?) echo "$USAGE" ;
            exit 1
            ;;
    esac
done
```

## Using getopt IV

- Now that you have dealt with option parsing, you need to deal with still other error conditions.
- For example, what should your script do if the input file is not specified?
- The simplest answer would be to exit with an error, but with a little more work, you can make the script much more user-friendly.
- If you use the fact that **getopts** sets the variable OPTIND to the value of the last option that it scanned, you can have the script assume that the first argument after this is the input filename. If no additional arguments remain, you should exit.
- Your error checking consists of the following lines:

## Using getopt V

```
shift `echo "$OPTARG - 1" | bc`  
if [ -z "$1" -a -z "$INFILE" ] ; then  
    echo "ERROR: Input file was not specified."  
    exit 1  
fi  
if [ -z "$INFILE" ] ; then INFILE="$1" ; fi
```

- Here you use the **shift** command to discard the arguments given to the script by one minus the last argument processed by getopt. The exact number of arguments to shift is calculated by the bc command, which is a command line calculator.
- Strictly speaking, you do not have to shift the arguments. It simplifies the if statement.



## Using getopt VI

- After shifting the arguments, check whether the new \$1 contains some value. If it does not, print and exit. Otherwise, set INFILE to the filename specified by \$1.
- You also need to set the output filename, in case the -o option was not specified. You can use variable substitution to accomplish this:

```
: ${OUTFILE:=${INFILE}.uu}
```

- Here the name of the output file is set to the input file plus the .uu extension, if an output file is not given.
- Note that you use the : command to prevent the shell from trying to execute the result of the variable substitution.
- When you have made sure that all the inputs are correct, the actual work is quite simple. The uuencode command that you use is:

## Using getopt VII

```
uuencode $INFILE $INFILE > $OUTFILE ;
```

- You should also check whether the input file is really a file before doing this command, so the actual body is:

```
if [ -f "$INFILE" ] ; then  
    uuencode $INFILE $INFILE > $OUTFILE;  
fi
```

- At this point the script is fully functional, but you still need to add the verbose reporting. This changes the preceding if statement to the following:

## Using getopt VIII

```
if [ -f "$INFILE" ] ; then
    if [ "$VERBOSE" = "true" ] ; then
        echo "uencoding $INFILE to $OUTFILE..."
    fi
    uuencode $INFILE $INFILE > $OUTFILE ; RET=$? ;
    if [ "$VERBOSE" = "true" ] ; then
        MSG="Failed" ;
        if [ $RET -eq 0 ] ; then MSG="Done." ; fi
        echo $MSG
    fi
fi
```

- See example *uu*.

# Using getopts IX

- With this script you can uuencode files in all of the following ways (assuming the script is called uu):
  - `./uu ch11.doc`
  - `./uu -f ch11.doc`
  - `./uu -f ch11.doc -o ch11.uu`