Working With Options

Processing simple options

```
#!/bin/bash
# extracting command line options as parameters
while [ -n "$1" ]
do
case "$1" in
-a) echo "Found the -a option" ;;
-b) echo "Found the -b option";;
-c) echo "Found the -c option" ;;
*) echo "$1 is not an option";;
esac
shift
done
$ ./test15 -a -b -c -d
```

Separating options from parameters

- The special character is the double dash (--). The shell uses the double dash to indicate the end of the option list.

Example

```
while [ -n "$1" ]
do
case "$1" in
-a) echo "Found the -a option" ;;
-b) echo "Found the -b option";;
-c) echo "Found the -c option" ;;
--) shift
break ;;
*) echo "$1 is not an option";;
esac
shift
done
count=1
for param in $@
do
echo "Parameter #$count: $param"
count=$[ $count + 1 ]
done
```

Getting User Input

Basic reading

```
#!/bin/bash
# testing the read command
echo -n "Enter your name: "
read name
echo "Hello $name, welcome to my program."
```

Reading Multiple Variables

```
#!/bin/bash
# entering multiple variables
read -p "Enter your name: " first last
echo "Checking data for $last, $first..."
```


REPLY

```
#!/bin/bash
# testing the REPLY environment variable
read -p "Enter a number: "
factorial=1
for (( count=1; count <= $REPLY; count++ ))
do
factorial=$[ $factorial * $count ]
done
echo "The factorial of $REPLY is $factorial"
```

Timing out

- you can use the -t option specify a timer.

```
#!/bin/bash
# timing the data entry
if read -t 5 -p "Please enter your name: " name
then
echo "Hello $name, welcome to my script"
else
echo
echo "Sorry, too slow!"
fi
```


Silent reading

- The -s option prevents the data entered in the read command from being displayed on the monitor.

```
#!/bin/bash
# hiding input data from the monitor
read -s -p "Enter your password: " pass
echo
echo "Is your password really $pass?"
```

Reading from a file

-
- You can also use the read command to read data stored in a file on the Linux system.
- Each call to the read command reads a single line of text from the file.
- This can done using a Pipe.

Example

```
#!/bin/bash
# reading data from a file
count=1
cat test | while read line
do
echo "Line $count: $line"
count=$[ $count + 1]
done
echo "Finished processing the file"
```

PRESENTING DATA

Understanding Input and Output

- So far, you've seen two methods for displaying the output from your scripts:
  - Display output on the monitor screen
  - Redirect output to a file

Standard file descriptors

- The Linux system handles every object as a file. This includes the input and output process.
- Linux identifies each file object using a file descriptor.

Linux Standard File Descriptors

File Descriptor	Abbreviation	Description
0	STDIN	Standard Input
1	STDOUT	Standard Output
2	STDERR	Standard Error

STDIN

- The STDIN file descriptor references the standard input to the shell.
- When you use the input redirect symbol (<), Linux replaces the standard input file descriptor with the file referenced by the redirection.

Example

```
$ cat
this is a test
this is a test
this is a second test.
this is a second test.
```

When you enter the cat command on the command line by itself, it accepts input from STDIN.
As you enter each line, the cat command echoes the line to the display.

Redirecting To A File

- However, you can also use the STDIN redirect symbol to force the cat command to accept input from another file other than STDIN:

```
$ cat < testfile
This is the first line.
This is the second line.
This is the third line.
$
```


STDOUT

- The STDOUT file descriptor references the standard output for the shell.
- Most bash commands direct their output to the STDOUT file descriptor by default. You can change that using output redirection:

```
$ ls -l > test2
$ cat test2
```

STDERR

- The shell handles error messages using the special STDERR file descriptor.
- The STDERR file descriptor references the standard error output for the shell.
- By default, the STDERR file descriptor points to the same place as the STDOUT file descriptor.

Redirecting just errors

- The STDERR file descriptor is set to the value 2.
- You can select to redirect only error messages by placing this file descriptor value immediately before the redirection symbol.

```
$ ls -al badfile 2> test4
$ cat test4
ls: cannot access badfile: No such file or directory
$
```

- Mixing STDOUT and STDERR messages in the same output:
  - $ ls -al test badtest test2 2> test5

Redirecting errors and data

- If you want to redirect both errors and the normal output, you'll need to use two redirection symbols.
  - $ ls -al test test2 test3 badtest 2> test6 1> test7

Redirecting to the Same File

- Alternatively, if you want, you can redirect both STDERR and STDOUT output to the same output file.
- The bash shell provides a special redirection symbol just for this purpose, the &> symbol:
  - $ ls -al test test2 test3 badtest &> test7

REDIRECTING OUTPUT IN SCRIPTS

Two Ways

- There are two methods for redirecting output in the script:
  - Temporarily redirecting each line
  - Permanently redirecting all commands in the script

Temporary redirections

- If you want to purposely generate error messages in your script, you can redirect an individual output line to STDERR.
- When you redirect to a file descriptor, you must precede the file descriptor number with an ampersand sign (&):
  - echo "This is an error message" >&2

Example

```
#!/bin/bash
# testing STDERR messages
echo "This is an error" >&2
echo "This is normal output"
$
```

- Its better to redirect to a file
  - $ ./test8 2> test9

Permanent redirections

- you can tell the shell to redirect a specific file descriptor for the duration of the script by using the exec command:

```
#!/bin/bash
# redirecting all output to a file
exec 1>testout
echo "This is a test of redirecting all output"
echo "from a script to another file."
echo "without having to redirect every individual
  line"
```

You can also redirect the STDOUT in the middle of a script

```
#!/bin/bash
# redirecting output to different locations
exec 2>testerror
echo "This is the start of the script"
echo "now reidirecting all output to another location"
exec 1>testout
echo "This output should go to the testout file"
echo "but this should go to the testerror file" >&2
```


Redirecting Input in Scripts

- You can use the same technique used to redirect STDOUT and STDERR in your scripts to redirect STDIN from the keyboard.
- The exec command allows you to redirect STDIN to a file on the Linux system:
  - exec 0< testfile

Example

```
#!/bin/bash
# redirecting file input
exec 0< testfile
count=1
while read line
do
echo "Line #$count: $line"
count=$[ $count + 1 ]
done
```


Creating output file descriptors

```
#!/bin/bash
# using an alternative file descriptor
exec 3>test13out
echo "This should display on the monitor"
echo "and this should be stored in the file" >&3
echo "Then this should be back on the monitor"
```

Redirecting file descriptors

```
#!/bin/bash
# storing STDOUT, then coming back to it
exec 3>&1
exec 1>test14out
echo "This should store in the output file"
echo "along with this line."
exec 1>&3
echo "Now things should be back to normal"
```


Creating input file descriptors

- You can redirect input file descriptors exactly the same way as output file descriptors.

```
#!/bin/bash
# redirecting input file descriptors
exec 6<&0
exec 0< testfile
count=1
while read line
do
echo "Line #$count: $line"
count=$[ $count + 1 ]
done
exec 0<&6
read -p "Are you done now? " answer
case $answer in
Y|y) echo "Goodbye";;
N|n) echo "Sorry, this is the end.";;
esac
```

Creating a read/write file descriptor

```
#!/bin/bash
# testing input/output file descriptor
exec 3<> testfile
read line <&3
echo "Read: $line"
echo "This is a test line" >&3
```

- *This will not produce the desired output. Since an internal pointer is maintained when reading file. The line will be written starting at the current position of the pointer*

Closing file descriptors

- The shell automatically closes the file descriptor.
- You can also close it manually.
- To close a file descriptor, redirect it to the special symbol &-. This is how this looks in the script:
  - exec 3>&-

Example

```
#!/bin/bash
# testing closing file descriptors
exec 3> test17file
echo "This is a test line of data" >&3
exec 3>&-
echo "This won't work" >&3
```


A Word of Caution

- There's yet another thing to be careful of when closing file descriptors.
- If you open the same output file later on in your script, the shell replaces the existing file with a new file.

Example

```
#!/bin/bash
# testing closing file descriptors
exec 3> test17file
echo "This is a test line of data" >&3
exec 3>&-
cat test17file
exec 3> test17file
echo "This'll be bad" >&3
```


Listing Open File Descriptors

- The lsof command lists all of the open file descriptors on the entire Linux system.
- the lsof command is located in the /usr/sbin directory.
- You can run it like this:
  - $ /usr/sbin/lsof

Command Line Options and Parameters

- -p, which allows you to specify a process ID (PID).
- -d, which allows you to specify the file descriptor numbers to display.
- $$, PID of the current process.
- The -a option is used to AND the results of the other two options.
- Example:
  - $ /usr/sbin/lsof -a -p $$ -d 0,1,2

Default lsof output

Column	Description
COMMAND	The first nine characters of the name of the command in the process
PID	The process ID of the process
USER	The login name of the user who owns the process
FD	The file descriptor number and access type (r - read, w - write, u - read/write)
TYPE	The type of file (CHR - character, BLK - block, DIR - directory, REG - regular file)
DEVICE	The device numbers of the device
SIZE	If available, the size of the file
NODE	The node number of the local file
NAME	The name of the file

Suppressing Command Output

- There are times when you don't want to display any output from your script.
- To do that you use a special file called the *null file.*
- This file is located at:
  - /dev/null
  - Any data you redirect to that location is thrown away and doesn't appear:
    - $ ls -al > /dev/null
    - $ cat /dev/null

Suppressing Errors

- This is a common way to suppress any error messages without actually saving them:
  - $ ls -al badfile test16 2> /dev/null

USING TEMPORARY FILES

What are temp files ?

- The Linux system contains a special directory location reserved for temporary files.
- Linux uses the /tmp directory for files that don't need to be kept indefinitely.

Creating a local temporary file

- By default, mktemp creates a file in the local directory.
- All you need to do is specify a filename template.
- The template consists of any text filename, plus six X's appended to the end of the filename:
  - $ mktemp testing.XXXXXX

Using it in Scripts

```
#!/bin/bash
# creating and using a temp file
tempfile=`mktemp test19.XXXXXX`
exec 3>$tempfile
echo "This script writes to temp file $tempfile"
echo "This is the first line" >&3
echo "This is the second line." >&3
echo "This is the last line." >&3
exec 3>&-
echo "Done creating temp file. The contents are:"
cat $tempfile
```

Creating a temporary file in /tmp

- The -t option forces mktemp to create the file in the temporary directory of the system.
  - $ mktemp -t test.XXXXXX

Creating a temporary directory

- The -d option tells the mktemp command to create a temporary directory instead of a file.

```
#!/bin/bash
# using a temporary directory
tempdir=`mktemp -d dir.XXXXXX`
cd $tempdir
tempfile1=`mktemp temp.XXXXXX`
tempfile2=`mktemp temp.XXXXXX`
exec 7> $tempfile1
exec 8> $tempfile2
echo "Sending data to directory $tempdir"
echo "This is a test line of data for $tempfile1" >&7
```

Logging Messages

- Sometimes it's beneficial to send output both to the monitor and to a file for logging.
- Instead of having to redirect output twice, you can use the special tee command.
  - $ date | tee testfile

Appending Data

- If you want to append data to the file, you must use the -a option:
  - $ date | tee -a testfile