28

Answers to Even-Numbered Exercises

2. The special parameter "**$@**" is referenced twice in the **out** script (page 883). Explain what would be different if the parameter "**$***" were used in its place.

   If you replace "**$@**" with "**$***" in the **out** script, cat or less would be given a single argument: a list of all files you specified on the command line enclosed within single quotation marks. This list works when you specify a single filename. When you specify more than one file, the shell reports **No such file or directory** because there is not a file named the string you specified on the command line (the SPACEs are not special characters when they are enclosed within single quotation marks).

4. Write a function that takes a single filename as an argument and adds execute permission to the file for the user.

```
$ function perms () {
> chmod u+x $1
> }
```

   a. When might such a function be useful?

      When you are writing many shell scripts, it can be tedious to give many chmod commands. This function speeds up the process.

   b. Revise the script so that it takes one or more filenames as arguments and adds execute permission for the user for each file argument.

```
$ function perms {
> chmod u+x $*
> }
```

   c. What can you do to make the function available every time you log in?

      Put the function in **~/.bash_profile** and/or **~/.bashrc** to make it available each time you log in (using bash).

d. Suppose that, in addition to having the function available on subsequent login sessions, you want to make the function available now in your current shell. How would you do so?

Use source to execute the file you put the function in, for example,

```
$ source ~/.bash_profile
```

6. Write a shell script that displays the names of all directory files, but no other types of files, in the working directory.

There are many ways to solve this problem. The **listdirs** script uses file to identify directory files and grep to pull them out of the list. Then sed removes everything from file's output, starting with the colon.

```
$ cat listdirs
file "$@"      |
grep directory |
sed 's/:.*//'
```

8. Enter the following script named **savefiles**, and give yourself execute permission to the file:

```
$ cat savefiles
#! /bin/bash
echo "Saving files in current directory in file savethem."
exec > savethem
for i in *
        do
        echo "================================================="
        echo "File: $i"
        echo "================================================="
        cat "$i"
        done
```

a. What error message do you get when you execute this script? Rewrite the script so that the error does not occur, making sure the output still goes to **savethem**.

You get the following error message:

```
cat: savethem: input file is output file
```

Add the following lines after the line with **do** on it:

```
if [ $i == "savethem" ] ; then
        continue
fi
```

b. What might be a problem with running this script twice in the same directory? Discuss a solution to this problem.

Each time you run **savefiles**, it overwrites the **savethem** file with the current contents of the working directory. When you remove a file and run **savefiles** again, that file will no longer be in **savethem**. If you want to

keep an archive of files in the working directory, you need to save the files to a new file each time you run **savefiles**. If you prefix the filename **savethem** with **$$**, you will have a unique filename each time you run **savefiles**.

10. Using the find utility, perform the following tasks:

    a. List all files in the working directory and all subdirectories that have been modified within the last day.

    ```
    $ find . -mtime -1
    ```

    b. List all files that you have read access to on the system that are larger than 1 megabyte.

    ```
    $ find / -size +1024k
    ```

    c. Remove all files named **core** from the directory structure rooted at your home directory.

    ```
    $ find ~ -name core -exec rm {} \;
    ```

    d. List the inode numbers of all files in the working directory whose filenames end in **.c**.

    ```
    $ find . -name "*.c" -ls
    ```

    e. List all files that you have read access to on the root filesystem that have been modified in the last 30 days.

    ```
    $ find / -xdev -mtime -30
    ```

12. Write a script that takes the name of a directory as an argument and searches the file hierarchy rooted at that directory for zero-length files. Write the names of all zero-length files to standard output. If there is no option on the command line, have the script delete the file after displaying its name, asking the user for confirmation, and receiving positive confirmation. A **–f** (force) option on the command line indicates that the script should display the filename but not ask for confirmation before deleting the file.

    The following script segment deletes only ordinary files, not directories. As always, you must specify a shell and check arguments.

    ```
    $ cat zerdel
    if [ "$1" == "-f" ]
            then
                    find $2 -empty -print -exec rm -f {} \;
            else
                    find $1 -empty -ok rm -f {} \;
    fi
    ```

14. Generalize the script written in exercise 13 so that the character separating the list items is given as an argument to the function. If this argument is absent, the separator should default to a colon.

    This script segment takes an optional option in the form **–d***x* to specify the delimiter ***x***:

```
$ cat nodel
if [[ $1 == -d? ]]
        then
                del=$(echo $1 | cut -b3)
                shift
        else
                del=:
fi
IFS=$del
set $*
for i
        do
        echo $i
done
```

16. Rewrite **bundle** (page 910) so that the script it creates takes an optional list of filenames as arguments. If one or more filenames are given on the command line, only those files should be re-created; otherwise, all files in the shell archive should be re-created. For example, suppose that all files with the filename extension **.c** are bundled into an archive named **srcshell**, and you want to unbundle just the files **test1.c** and **test2.c**. The following command will unbundle just these two files:

```
$ bash srcshell test1.c test2.c
```

```
$ cat bundle2
#!/bin/bash
# bundle: group files into distribution package

echo "# To unbundle, bash this file"
for i
do
        echo 'if echo $* | grep -q' $i '|| [ $# = 0 ]'
                echo then
                echo "echo $i 1>&2"
                echo "cat >$i <<'End of $i'"
                cat $i
                echo "End of $i"
        echo fi
done
```

18. In principle, recursion is never necessary. It can always be replaced by an iterative construct, such as **while** or **until**. Rewrite **makepath** (page 950) as a nonrecursive function. Which version do you prefer? Why?

```
function makepath2()
{
wd=$(pwd)
pathname=$1

while [[ $pathname = */* && ${#pathname} > 0 ]]
        do
        if [[ ! -d $pathname ]]
                then
                mkdir "${pathname%%/*}"
        fi
        cd "${pathname%%/*}"
        pathname="${pathname#*/}"
        done
if [[ ! -d $pathname && ${#pathname} > 0 ]]
        then
        mkdir $pathname
fi
cd $wd
}
```

The recursive version is simpler: There is no need to keep track of the working directory and you do not have to handle making the final directory separately.

20. Write a function that takes a directory name as an argument and writes to standard output the maximum of the lengths of all filenames in that directory. If the function's argument is not a directory name, write an error message to standard output and exit with nonzero status.

```
$ function maxfn () {
> typeset -i max thisone
> if [ ! -d "$1" -o $# = 0 ]
>     then
>     echo "Usage: maxfn dirname"
>     return 1
> fi
>
> max=0
> for fn in $(/bin/ls $1)
>     do
>     thisone=${#fn}
>     if [ $thisone -gt $max ]
>         then
>         max=$thisone
>     fi
>     done
> echo "Longest filename is $max characters."
> }
```

22. Write a function that lists the number of ordinary files, directories, block special files, character special files, FIFOs, and symbolic links in the working directory. Do this in two different ways:

a. Use the first letter of the output of **ls –l** to determine a file's type.

```
$ function ft () {
 > typeset -i ord=0 dir=0 blk=0 char=0 fifo=0 symlnk=0 other=0
 >
 > for fn in *
 >     do
 >     case $(ls -ld "$fn" | cut -b1) in
 >         d)
 >             ((dir=$dir+1))
 >             ;;
 >         b)
 >             ((blk=$blk+1))
 >             ;;
 >         c)
 >             ((char=$char+1))
 >             ;;
 >         p)
 >             ((fifo=$fifo+1))
 >             ;;
 >         l)
 >             ((symlnk=$symlnk+1))
 >             ;;
 >         a-z)
 >             ((other=other+1))
 >             ;;
 >         *)
 >             ((ord=ord+1))
 >             ;;
 >     esac
 >     done
 >
 > echo $ord ordinary
 > echo $dir directory
 > echo $blk block
 > echo $char character
 > echo $fifo FIFO
 > echo $symlnk symbolic link
 > echo $other other
 > }
```

b. Use the file type condition tests of the *[[**expression**]]* syntax to determine a file's type.

```
$ function ft2 () {
 > typeset -i ord=0 dir=0 blk=0 char=0 fifo=0 symlnk=0 other=0
 >
 > for fn in *
 >     do
 >         if [[ -h $fn ]]
 >             then ((symlnk=$symlnk+1))
 >         elif [[ -f $fn ]]
 >             then ((ord=ord+1))
 >         elif [[ -d $fn ]]
 >             then ((dir=$dir+1))
 >         elif [[ -b $fn ]]
 >             then ((blk=$blk+1))
 >         elif [[ -c $fn ]]
 >             then ((char=$char+1))
 >         elif [[ -p $fn ]]
 >             then ((fifo=$fifo+1))
 >         else
 >             ((other=other+1))
 > fi
 >     done
 >
 > echo $ord ordinary
 > echo $dir directory
 > echo $blk block
 > echo $char character
 > echo $fifo FIFO
 > echo $symlnk symbolic link
 > echo $other other
 > }
```