CHAPTER W29

Introductory TC Shell Programming

Objectives

- To introduce the concept of shell programming
- To describe how TC shell programs can be executed
- To explain the concept and use of shell variables in TC shell
- To discuss how command-line arguments are passed to TC shell programs
- To discuss the concept of command substitution
- To describe some basic coding principles
- To write and discuss a few sample TC shell scripts
- To cover the commands and primitives

  *, =, ", ', `, &, <, >, ;, |, \, /, [], (), continue, tcsh, exit, env, foreach, goto, head, if, ls, set, setenv, shift, switch, while, unset, unseten

W29.1 INTRODUCTION

The TC shell is more than a command interpreter; it has a programming language of its own that can be used to write shell programs for performing various tasks that cannot be performed by any existing command. Shell programs, commonly known as *shell scripts*, in the TC shell consist of shell commands to be executed by a shell and are stored in ordinary Linux files. The shell allows the use of read/write storage places, called *shell variables*, to make it easier for the user or programmer to work and for programmers to use as scratch pads for completing tasks. The TC shell also has program control flow commands/statements that allow the writers of shell scripts to implement multiway branching and repeated execution of a block of commands.

W29.2 RUNNING A TC SHELL SCRIPT

There are three ways to run a TC shell script. The first step for all three methods is to make the script file executable by adding the execute permission to the existing access permissions for the file. You can do so by running the `chmod u+x hello_world` command, where **hello_world** is the name of the file containing the shell script. Without making the script file executable, TC shell displays the error message that the script in the **hello_world** file cannot be executed.

```
% cat hello_world
echo "Hello, world!"
$ ./hello_world
-bash: ./hello_world: Permission denied
$ chmod u+x hello_world
$
```

Clearly, in this case, you make the script executable for yourself only. However, you can set appropriate access permissions for the file if you also want other users to be able to execute it. Once you have made the script file executable, you can type `./hello_world` as a command to execute the shell script, as shown:

```
% ./hello_world
Hello, world!
%
```

If your search path (the `path` variable) includes your current directory (.), you can simply use the `hello_world` command, instead of using the `./hello_world` command. For the rest of this chapter, we assume that your `path` variable includes your current directory.

As described in Chapter 10, a child of the current shell process executes the script. Thus, with this method, the script executes properly if you are using TC shell but not if you are using any other shell. If you are currently using some other shell, first execute the `/bin/tcsh` command to run TC shell and then run the `hello_world` command, as shown in the following example. Here, we assume that your current shell is Bash (with the $ prompt). After the script has completed its execution, we press `<Ctrl+D>` to terminate bash and return to Bash.

```
$ /bin/tcsh
% hello_world
Hello, world!
% <Ctrl+D>
$
```

The second method of executing a shell script is to run the `/bin/tcsh` command with the script file as its parameter. Thus, the following command executes the shell script in **hello_world** even though the current shell is Bash.

```
$ /bin/tcsh hello_world
Hello, world!
$
```

If your `path` variable includes the **/bin** directory, you can simply use the `tcsh` command instead of using the `/bin/tcsh` command.

The third method, which is also the most commonly used method, is to force the current shell to execute a script in TC shell, regardless of your current shell. You can do so by beginning a shell script with the line containing `#!/bin/tcsh`, as shown in the following session:

```
$ cat hello_world
#!/bin/tcsh
echo "Hello, world!"
```

```
$
```

When your current shell encounters the string `#!`, it takes the rest of the line as the absolute pathname for the shell to be executed, under which the script in the file is executed.

Throughout this chapter, we would use the `chmod u+x script_file` command to make **script_file** executable by the owner of the file and run the script by using the `./script_file` command.

W29.3 Shell Variables and Related Commands

A *variable* is a main memory location that is given a name. This allows you to reference the memory location by using its name instead of its address. The name of a shell variable is comprised of digits, letters, and underscores, with the first character being a letter or underscore. Because the main memory is read/write storage, you can read a variable's value or assign it a new value. Under the TC shell, the value of a variable can be a string of characters or a numeric value. There is no theoretical limit to the length of a variable's value stored as a string; the length of a line dictates the practical limit.

Shell variables can be one of two types: *shell environment variables* and *user-defined variables*. You can use environment variables to customize the environment in which your shell runs and for proper execution of shell commands. A copy of these variables is passed to every command that executes in the shell as its child. Most of these variables are initialized when the login script(s) execute(s), according to the environment set by your system administrator. The **/etc/csh.cshrc** and **/etc/csh.login** are the systemwide **.login** and **.cshrc** files for `tcsh`. A login `tcsh` shell starts by executing commands in these files. You can further customize your environment by assigning appropriate values to some or all of these variables, as well as define additional variables, in your **~/.tcshrc**, **~/.cshrc**, and **~/.login** start-up files, which also execute if `tcsh` is your login shell. See Chapter 2 for a discussion of start-up files. Table W29.1 lists most of the environment variables whose values you can change.

TABLE W29.1 Some Important Writable TC Shell Environment Variables

Environment Variable	Purpose of the Variable
`cdpath`	Directory names that are searched, one by one, by the `cd` command to find the directory passed to it as a parameter; the `cd` command searches the current directory if this
`home`	Name of your home directory, the TC shell places you in this directory when you first log
`mail`	Name of your system mailbox file
`path`	Variable that contains your search path—the directories that a shell searches to find an
`prompt`	Primary shell prompt that appears on the command line, usually set to `%`
`prompt2`	Secondary shell prompt displayed on the second line of a command if the shell thinks that the typed command is not complete, typically when the command line terminates with a
`cwd`	Name of the current working directory
`term`	Type of user's console terminal

The shell environment variables listed in Table W29.1 are *writable*, meaning that you can assign them any values to make your shell environment meet your needs. Other shell environment variables are *read only*. That is, you can use (read) the values of these variables, but you cannot change them directly. These variables are most useful for processing command-line arguments (also known as *positional arguments*), the parameters passed to a shell script at the command line. Examples of command-line arguments are the source and destination files in the `cp` command. Some other read-only shell variables are used to keep track of the process ID of the current process, the process ID of the most recent background process, and the exit status of the last command. Some important *read-only shell environment variables* are listed in Table W29.2.

TABLE W29.2 Some Important Read-Only TC Shell Environment Variables

Environment Variable	**Purpose of the Variable**
`$argv[0]` or `$0`	Name of the executing program
`$number or ${number}`	Equivalent to `$argv[number]`, where the number may be 0, 1, 2,
`$argv[*] or $*`	Values of all of the command-line arguments
`$#argv or $#`	Total number of command-line arguments
`$$`	Process ID (PID) of the current process; typically used as a file
`$!`	PID of the most recent background process

User-defined variables are used within shell scripts as temporary storage places whose values can be changed when the program executes. These variables can be made global and passed to the commands that execute in the shell script in which they are defined. As with most programming languages, you have to declare TC shell variables before using them. A reference to a TC shell variable that has not been declared, results in an error.

You can display the values of all shell variables (environmental and user-defined) and their current values by using the `set` command without any argument. The following is a sample run of the `set` command on our

Linux machine.

```
% set
_

addsuffix
anyerror
argv ()
autoexpand
autolist
csubstnonl
cwd /home/sarwar
dirstack   /home/sarwar
echo_style both
edit
euid1004
euser      sarwar
gid 1002
group      faculty
history    100
home /home/sarwar
killring   30
owd
path (/usr/local/sbin /usr/local/bin /usr/sbin /usr/bin /sbin /bin /usr/games
/usr/local/games)
... [output truncated] ...
%
```

The `@` command displays the same information as the `set` command does. You can use the `env` and `printenv` commands to display both the environment variables and their values. The following is a sample output of the env command on the same Linux machine that we ran the set command on.

```
% env
LC_PAPER=ur_PK
LC_ADDRESS=ur_PK
XDG_SESSION_ID=117
LC_MONETARY=ur_PK
TERM=xterm-color
SHELL=/bin/bash
XDG_SESSION_COOKIE=4603ef0b0eb640e2836974a04d12e5ab-1532888827.861531-827693703
SSH_CLIENT=103.255.4.40 57146 22
LC_NUMERIC=ur_PK
SSH_TTY=/dev/pts/0
USER=sarwar
...
MAIL=/var/mail/sarwar
PATH=/home/sarwar/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/us
r/games:/usr/local/games:.
LC_IDENTIFICATION=ur_PK
PWD=/home/sarwar
LANG=en_US.UTF-8
LC_MEASUREMENT=ur_PK
SHLVL=2
HOME=/home/sarwar
LOGNAME=sarwar
SSH_CONNECTION=103.255.4.40 57146 202.147.169.195 22
LESSOPEN=| /usr/bin/lesspipe %s
XDG_RUNTIME_DIR=/run/user/1004
LESSCLOSE=/usr/bin/lesspipe %s %s
LC_NAME=ur_PK
```

```
_=/usr/bin/tcsh
HOSTTYPE=x86_64-linux
VENDOR=unknown
OSTYPE=linux
MACHTYPE=x86_64
GROUP=faculty
HOST=Mint18
REMOTEHOST=103.255.4.40
%
```

In the following in-chapter exercises, you will create a simple shell script and make it executable. Also, you will use the `set` and `env` commands to display the names and values of shell variables in your environment.

Exercise W29.1

Display the names and values of all shell variables on your Linux machine. What command(s) did you use?

Exercise W29.2

Create a file that contains a TC shell script comprising the `date` and `who` commands, one on each line. Make the file executable and run the shell script. List all the steps for completing this task.

W29.4 Reading and Writing Shell Variables

You can use any of three commands to assign a value to (write) one or more shell variables (environmental or user-defined): `@`, `set`, and `setenv`. The `set` and `setenv` commands are used to assign a string to a variable. The difference is that the `setenv` command declares and initializes a global variable (equivalent to an assignment statement followed by execution of the `export` command in Bash), whereas the `set` command declares and initializes a local variable. Also, with the `set` command you can declare and initialize multiple variables in a command line, whereas you can do so with only one variable in the case of `setenv` command. You can use the `@` command to assign an integer value to a local variable. The following are brief descriptions of the `@` and `set` commands. We describe the `setenv` command in Section W29.4.2.

SYNTAX

```
set [variable1[=strval1] variable2 [=strval2] … variableN [=strvalN]]

@ [variable1[=numval1] variable2 [=numval2] … variableN [=numvalN]]
```

Purpose: Assign values **`strval1`**, …, **`strvalN`** or **`numval1`**, …, **`numvalN`** to variables **`variable1`**, …, **`variableN`**, respectively, where a value can be **`strval`** for a string value and **`numval`** for a numeric value

No space is required before or after the = sign for the `@` and `set` commands, but spaces can be used for clarity. If a value contains spaces, you must enclose the value in parentheses. The `set` command with only the name of a variable declares the variable and assigns it a null value. Unlike the Bourne shell, where every variable is automatically initialized, in the TC shell you must declare a variable to initialize and use it. Without any arguments, the `set` and `@` commands display all shell variables and their values. Multiword values are displayed in parentheses. We discuss the `@` command in detail in Chapter W30.

You can refer to (i.e., read) the current value of a variable by inserting a dollar sign (`$`) before the name of a variable, as in `$variable`. You can use `${variable}` in place of `$variable` to insulate "variable" from the characters that follow it. You can use `${?variable}` to check whether "variable" is set or not; it returns 1 if "variable" is set and 0 if it isn't. You can use the `echo` command to display values of shell variables.

In the following session, we show how shell variables can be read and written. We also show the use of single and double quotes, *, and \ in an assignment statement.

```
% echo $name
name: Undefined variable.
% set name
% echo $name

% echo ${?name}
1
% set name = John
% echo $name
John
% echo ${?name}
1
% echo ${?place}
0
% set name = John Doe II
% echo $name
John
% echo $Doe $II

% set name = (John Doe)
% echo $name
John Doe
% set name = John*
% echo $name
John.Bates.letter   John.Johnsen.memo   John.email
% set name = "John*"
% echo "$name"
John*
% echo "The name $name sounds familiar!"
The name John* sounds familiar!
```

```
% echo \$name
$name
% echo '$name'
$name
%
```

The preceding session shows that, if values that include spaces are not enclosed in quotes, the shell assigns the first word to the variable and the remaining as null-initialized variables. In other words, the command `set name = John Doe II` initializes the `name` variable to John, and declares `Doe` and `II` as string variables initialized to null. When you display a variable initialized to null, the shell displays a blank line. Also, after the `set name=John*` command has been executed and `$name` is not enclosed in quotes in the `echo` command, the shell lists the file names in your present working directory that match `John*`, with `*` considered as the shell metacharacter—that is, files that start with the string `John`, followed by 0 or more characters. If your current directory does not contain any files that start with the string `John`, the `set name = John*` command returns an error message, `set: No match.`, and a subsequent `echo $name` command would display the previous value of the name variable, `John Doe` in this case. Running the `echo *` command would display the names of all the files in your current directory. The preceding session also shows that single quotes can be used to process the value of the `name` variable literally. In fact, you can use single quotes to process the whole string literally. The backslash character can be used to escape the special meaning of any single character, including `$`, and treat it literally. In the TC shell, in order to escape the special meaning of `!`, you must use a backslash (\) before the `!` symbol if it is followed by any character other than a space. The outputs of the `echo ${?name}` and `${?place}` commands show that the `name` variable was defined to null and John, and the `place` variable was undefined.

A command consisting of `$variable` alone results in the value of `variable` being executed as a shell command. If the value of `variable` comprises a valid command, the expected results are produced. If `variable` does not contain a valid command, the shell, as expected, displays an appropriate error message.

The following session illustrates this point with examples. The variable used in this session is `command`.

```
% set command = pwd
% $command
/home/sarwar/linux2e/ch14
% set command = hello
% $command
hello: Command not found.
%
```

W21.4.1 Command Substitution

When a command is enclosed in back quotes (also known as *grave accents*), the shell executes the command and substitutes the output of the command for the command (including back quotes). This process is referred to as command substitution. The following is a brief description of command substitution.

SYNTAX
`` `command` ``
Purpose: Execute **`command`** and substitute its output for **`` `command` ``**

The next session illustrates the concept. In the first assignment statement, the variable called `command` is assigned the value `pwd`. In the second assignment statement, the output of the `pwd` command is assigned to the `command` variable.

```
% set command = pwd
% echo "The value of command is: $command."
The value of command is: pwd.
```

```
% set command = `pwd`
% echo "The value of command is: $command."
The value of command is: /home/sarwar/linux2e/ch14.
%
```

Command substitution can be specified in any command. For example, in the following command line, the output of the `date` command is substituted for “`date`” before the `echo` command is executed.

```
% echo "The date and time are `date`."
The date and time are Mon Feb 20 21:10:45 PKT 2017.
%
```

The following in-chapter exercises are designed to reinforce the creation and use of shell variables and the concept of command substitution.

Exercise W29.3

Assign your full name to a shell variable `myname` and echo its value. How did you accomplish the task? Show your work.

Exercise W29.4

Assign the output of the command `echo "Hello, world!"` to the `myname` variable and then display the value of `myname`. List the commands that you executed to complete this task.

W29.4.2 Exporting Environment

When a variable is created, subsequent shells do not automatically know it. The `setenv` command passes the *value* of a variable to subsequent shells. Thus, when a shell script is called and executed from another shell script, it does not get automatic access to the variables defined in the original (caller) script unless they are explicitly made available to it. You can use the `setenv` command to assign a value to a string variable and pass the value of the variable to subsequent commands that execute as children of the script. Because all read/write shell environment variables are available to every command, script, and subshell, they are initialized by the `setenv` command. The following is a brief description of the `setenv` command.

SYNTAX

`setenv [variable [strval]]`

Purpose: Assigns to variable a string value **`strval`** and exports **`variable`** and a copy of its value so that it is available to every command executed from this point on

The following session shows a simple use of the `setenv` command. The `name` variable is initialized to `John Doe` and is exported to subsequent commands executed under a subshell of the current shell. Note that unlike the `set` command, the `setenv` command requires that you enclose multiword values in double quotes, not in parentheses. The first echo command displays `echo: No match.,` because the `name` variable may only be accessed under a subshell of the current shell. We run another TC shell under the current shell and use the `echo $name` command to display the value of the `name` variable. The `ps` commands are used to display the shell processes after the creation of the new TC shell and after it is terminated with `<Ctrl+D>`.

```
% setenv name "John Doe"
% echo $name
echo: No match.
% tcsh
```

```
$ ps
  PID TTY          TIME CMD
18400 pts/1    00:00:00 bash
18416 pts/1    00:00:00 tcsh
18755 pts/1    00:00:00 tcsh
18757 pts/1    00:00:00 ps
$ echo $name
John Doe
% <Ctrl+D>
% ps
  PID TTY          TIME CMD
18400 pts/1    00:00:00 bash
18416 pts/1    00:00:00 tcsh
18771 pts/1    00:00:00 ps
%
```

The next session illustrates the concept of exporting shell variables via some simple shell scripts.

```
% cat display_name
#!/bin/tcsh
echo $name
exit 0
% set name = (John Doe)
% ./display_name
name: Undefined variable.
%
```

Note that the script in the **display_name** file displays an undefined variable error message, even though we initialized the `name` variable just before executing this script. The reason is that the `name` variable declared interactively is not exported before running the script, and the `name` variable used in the script is local to the script. As this local variable `name` is uninitialized, the `echo` command displays the error message. As stated earlier, unlike the Bourne shell, the TC shell requires declaration of a variable before its use.

You can use the `exit` command to transfer control out of the executing program and pass it to the calling process, the current shell process in the preceding session. The only argument of the `exit` command is an optional integer number that is returned to the calling process as the exit status of the terminating process. All Linux commands return an exit status of zero upon *success*—that is, after successfully performing their tasks, and nonzero upon *failure*. The return status value of a command is stored in the read-only environment variable `$?` and can be checked by the calling process. In shell scripts, depending on the task at hand, the status of a command execution can be checked and then subsequent action can be taken. Later in the chapter, we show the use of the read-only environment variable `$?` in some shell scripts. When the `exit` command is executed without an argument, the Linux kernel sets the return status value for the script.

In the following session, the `name` variable is exported after it has been initialized, thus making it available to the `display_name` script. The session also shows that the return status of the `display_name` script is 0, implying successful execution of `display_name`.

```
% setenv name "John Doe"
% ./display_name
John Doe
% echo $?
0
%
```

We now show that a copy of an exported variable's value is passed to any subsequent command. That is, a command has access only to the value of an exported variable; it cannot assign a new value to the variable. Consider the following script in the **export_demo** file.

```
% cat export_demo
#!/bin/tcsh
setenv name "John Doe"
./display_change_name
echo "$name"
% cat display_change_name
#!/bin/tcsh
echo "$name"
set name = (Plain Jane)
echo "$name"
exit 0
% ./export_demo
John Doe
Plain Jane
John Doe
%
```

When the `export_demo` script is invoked, the `name` variable is set to `John Doe` and exported so that it becomes part of the environment of the commands that execute under `export_demo` as its children. The first `echo` command in the `display_change_name` script displays the value of the exported variable `name`. It then initializes a local variable called `name` to `Plain Jane`. The second `echo` command therefore echoes the current value of the local variable `name` and displays `Plain Jane`. When the `display_change_name` script finishes, the `display_name` script executes the `echo` command and displays the value of the exported `name` variable, thus displaying `John Doe`.

W29.4.3 Resetting Variables

A variable retains its value for as long as the script in which it is initialized executes. You can remove a variable from the environment by using the `unset` and `unsetenv` commands. The following are brief descriptions of the commands.

SYNTAX

```
unset variable-list

unsetenv variable
```

Purpose: Remove the specified variables from the environment. The variables in **variable-list** are separated by spaces. The **`unset`** command is used for the variables declared by the `set` or `@` commands. The **`unsetenv`** command is used for variables declared by the `setenv` command.

Next we show a simple use of the `unset` command. First, we undefined the `name` variable defined with the `setenv name` command in the earlier session. The variables `name` and `place` are set to `John` and `Corvallis`, respectively, and the `echo` command displays the values of these variables. The `unset` command resets the `name` variable, defined with the `set` command, to null. Thus, the `echo "$name"` command should display a message saying that the `name` variable is undefined. However, the command shows `John Doe` as its output. If you go back and look at the shell session before the last one, you would see that `name` was initialized to `John Doe` using the `setenv name "John Doe"` command. The `unset name` command in the following session undefined the `name` variable defined by using this command, but the `name` variable defined using the `setenv` command remained defined with the value `John Doe`. When we run the `unsetenv name` command in the following session, the `name` variable defined with the `setenv name "John Doe"` command is undefined and the `echo "$name"` command shows the error message "`name: Undefined variable.`".

```
% unsetenv name
% echo $name
name: Undefined variable.
% set name = John place = Corvallis
% echo $name $place
John Corvallis
% unset name
% echo $name
name: Undefined variable.
% echo $place
Corvallis
%
```

The following command removes from the environment the variables `name` and `place`, defined with the `set` command:

```
% unset name place
%
```

To reset a variable, explicitly assign it a null value by using the `set` command with the variable name only. Or you can assign the variable no value and simply hit `<Enter>` after the = sign, as in:

```
% set country=
% echo $country

% set place
% echo $place
```

```
%
```

Here, the `set` command is used to reset the `country` and `place` variables to null.

W29.4.4 Creating Read-Only User-Defined Variables

When programming, you need to use constants at times. You can use literal constants, but using symbolic constants is a good programming practice, primarily because it makes your code more readable. Another reason for using symbolic names is that a constant used at various places in code may need to be changed. With a symbolic constant, the change is made at one place only, but a literal constant must be changed every place it was used. A symbolic constant can be created in a TC shell by using the `set -r` command. The following is a brief description of the command.

SYNTAX

```
set -r [name[=value]]
```

Purpose: Declare read-only variables or set specified variables to read-only

In the following session, the `name` and `place` variables are read-only. Both variables are made read-only at the time of their creation. Once they have become read-only, assignment to either variable fails.

```
% set -r name = Jim place = Ames
% echo $name $place
Jim Ames
% set name = Art
set: $name is read-only.
```

```
% set place = (Ann Arbor)
set: $place is read-only.
%
```

When the set –r command is executed without arguments, it displays all user-defined read-only variables and their values, as shown in the following session.

```
% set -r
name        Jim
place       Ames
%
```

The environment variables cannot be made read-only.

W29.4.5 Reading from Standard Input

So far, we have shown how you can assign values to shell variables statically at the command line or by using the assignment statement. If you want to write an interactive shell script that prompts the user for keyboard input, you need to use the `set` command to store the user input in a shell variable, according to the following syntaxes.

SYNTAX

```
set variable = $<

set variable = "$<"
set variable = `echo $<`

set variable = `head -1`
```

Purpose: Read one line from **stdin** into **`variable`**; note that the use of back quotes (grave accents) in **`` `head -1` ``** that **`` `echo $<` ``**

These commands allow you to read one line of keyboard input into “variable.” The first syntax allows assignment of the first word of the input to “variable.” You can use the remaining three syntaxes for the command to assign the first line of keyboard input to “variable.” Unlike that of Bash, the keyboard input feature of the TC shell does not allow assignment of words in a line to multiple variables. However, the words in a line are stored in the form of an array, and you can access them using the name of the variable (we discuss arrays in Chapter W30).

We illustrate the semantics of the second `set` command with a script in the **keyin_demo** file, as follows:

```
% cat keyin_demo
#!/bin/tcsh
echo -n "Enter input: "
set line = `head -1`
echo "You entered: $line"
exit 0
%
```

The `` set line = `head -1` `` command can be replaced by the `set line = "$<"` or `` set line = `echo $<` `` command.

In the following run, enter `Linux rules the network computing world`! The `set` command takes the whole input and puts it in the shell variable `line` without the new line character. The output of the `echo` command displays the contents of the shell variable line.

```
% ./keyin_demo
Enter input: Linux rules the network computing world!
You entered: Linux rules the network computing world!
%
```

The -n option is used with the echo command to keep the cursor on the same line. If you do not use this option, the cursor moves to the next line, which is what you want to see happen while displaying information and error messages. However, when you prompt the user for keyboard input, you should keep the cursor in front of the prompt for a user-friendlier interface. The −n option in the echo command that prompts you for input can be replaced by \c at the end of the command, as in echo "Enter input: \c". The \c character, known as a special character, forces the cursor to stay at the same line after displaying Enter input: The TC shell echo command supports many other special characters. These characters, along with their meaning are listed in Table W29.3. Standard ASCII control sequences can be used to display other special characters, such as <Ctrl+H> for backspace and <Ctrl+G> for bell.

TABLE W29.3 Special Characters for the echo Command

Chatacter	Description
\b	Backspace
\c	Keep cursor on the same line
\f	Form feed
\n	New line (move cursor to next line)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash (escape special meaning of backslash)

`\0N`	Character whose ASCII number is N (in decimal)

In the following in-chapter exercise, you will use the `set` command to practice reading from **stdin** in shell scripts.

Exercise W29.5

Write commands for reading a value into the `myname` variable from the keyboard and exporting it so that the commands executed in any child shell have access to the variable.

Exercise W29.6

Try the shell sessions involving the `set -r` command on your system.

W29.5 PASSING ARGUMENTS TO SHELL SCRIPTS

In this section, we describe how command-line arguments can be passed to shell scripts and manipulated by them. As we discussed in Section W29.3, you can pass command-line arguments, also called *positional parameters*, to a shell script. The values of these arguments can be referenced by using the names `$argv[number],` where `number` may assume values 0, 1, 2, and so on, with `$argv[0]` referring to the program (or command) name. The positional parameters may also be specified by using the `$number` or `${number}` notation, such as `$0`, `$1`, `$2`, and so on, or `${0}`, `${1}`, `${2}`, and so on. The curly braces are used to isolate `number` from the character(s) that may follow. If a positional argument referenced in your script is not passed as an argument, it is initialized to a value of null. However, while displaying such arguments, the `$number` notation works fine, but the `$argv[number]` notation results in an error message, `argv: Subscript out of range.`, which simply means that you are indexing the `argv` array for an element that does not exist. You can use the names `$#argv` or `$#` to refer to the total number of arguments passed in an execution of the script. The names `$argv[*]`, `argv`, or `$*` refer to the values of all of the arguments. The names `$argv[0]` and `$0` refer to the name of the script file (i.e., the command name). In the following session, we use the shell script in the **cmdargs_demo** file to show how you can use these variables.

```
% cat cmdargs_demo
#!/bin/tcsh
echo "The command name is $0."
echo "The number of command line arguments is $#argv."
echo -n "The values of the command line arguments are: "
echo "$1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11 $12"
echo "Another way to display command line arguments is: $argv[*]"
echo "Yet another is: $*"
exit 0
% ./cmdargs_demo 1 2 3 4 5 6 7 8 9 10 11 12
The command name is cmdargs_demo.
The number of command line arguments is 12.
The values of the command line arguments are: 1 2 3 4 5 6 7 8 9 10 11 12
Another way to display command line arguments is: 1 2 3 4 5 6 7 8 9 10 11 12
Yet another is: 1 2 3 4 5 6 7 8 9 10 11 12
% ./cmdargs_demo One Two 3 Four 5 6
The command name is cmdargs_demo.
The number of command line arguments is 6.
The values of the command line arguments are: One Two 3 Four 5 6
Another way to display command line arguments is: One Two 3 Four 5 6
Yet another is: One Two 3 Four 5 6
%
```

The TC shell maintains as many as command-line arguments at a time as the command line length allows. You can write scripts that handle command-line arguments, one at a time, by shifting the arguments left by one argument. To do so, use the `shift` command. By default, this command shifts the command-line arguments to the left by one position, moving `$argv[2]` to `$argv[1]`, `$argv[3]` become `$argv[2]`, and so on. The first argument, `$argv[1]`, is shifted out. Once shifted, the arguments cannot be restored to their original values. More than one position can be shifted if specified as an argument to the command. The following is a brief description of the command.

SYNTAX

`shift [variable]`

Purpose: Shift the words in **`variable`** one position to the left; if no variable name is specified, the command-line arguments are assumed

The script in the **shift_demo** file shows the semantics of the `shift` command with the implicit variable, the command-line arguments. The `shift` command shifts the first argument out and the remaining arguments to the left by one position. The three `echo` commands are used to display the current values of program names, all positional arguments (`$#argv[*]`), and the values of the first three positional parameters, respectively. The results of execution of the script are obvious.

```
% cat shift_demo
#!/bin/tcsh
echo "The name of the program is $0."
echo "The arguments are: $argv[*]."
echo "The first three arguments are: $argv[1] $argv[2] $argv[3]."
shift
echo "The name of the program is $0."
echo "The arguments are: $argv[*]."
echo "The first three arguments are: $argv[1] $argv[2] $argv[3]"
exit 0
% ./shift_demo 1 2 3 4 5 6 7 8 9 10 11 12
The name of the program is shift_demo.
The arguments are: 1 2 3 4 5 6 7 8 9 10 11 12.
The first three arguments are: 1 2 3.
The name of the program is shift_demo.
The arguments are: 2 3 4 5 6 7 8 9 10 11 12.
The first three arguments are: 2 3 4
%
```

Using the set command with argv as its argument can alter the values of positional arguments. The most effective use of this command is in conjunction with command substitution. The following is a brief description of the command.

SYNTAX

set argv = [argument-list]

Purpose: Set values of the positional arguments to the arguments in **argument-list**

The following is a simple interactive use of the command. The date command is executed to show that the output has six fields. The set argv = `date` command sets the positional parameters according to the output of the date command. In particular, $argv[1] is set to Mon, $argv[2] to Feb, $argv[3] to 13, $argv[4] to 08:36:01, $argv[5] to PKT, and $argv[6] to 2017. The echo $argv[*] command displays the values of all positional arguments. The second echo command displays the date in a commonly used form.

```
% date
Mon Feb 13 08:35:24 PKT 2017
% set argv = `date`
% echo $argv[*]
Mon Feb 13 08:36:01 PKT 2017
% echo "$argv[2] $argv[3], $argv[6]"
Feb 13, 2017
%
```

The script in **set_demo** shows another use of the command. When the script is run with a file argument, it generates a line that contains the file name, the file's inode number, and the file size (in bytes). The `set` command is used to assign the output of `ls -il` command as the new values of the positional arguments `$argv[1]` through `$argv[9]`. We show the output of the `ls -il` command in case you do not remember the format of the output of this command.

```
% cat set_demo
#!/bin/tcsh
set filename = $argv[1]
set argv = `ls -il $filename`
set inode = $argv[1]
set perms = $argv[2]
set size = $argv[6]
echo "File Name:        $filename"
echo "Inode Number:     $inode"
echo "Permissions:      $perms"
echo "Size (bytes):     $size"
exit 0
% ./set_demo set_demo
File Name:        set_demo
```

```
Inode Number:     16517764
Permissions:      -rwxr-xr-x
Size (bytes):     265
% ls -il set_demo
16517764 -rwxr-xr-x 1 sarwar faculty 265 Feb 13 08:37 set_demo
%
```

In the following in-chapter exercises, you will use the `set` and `shift` commands to reinforce the use and processing of command-line arguments.

Exercise W29.7

Write a shell script that displays all command-line arguments, shifts them to the left by two positions, and redisplays them. Show the script along with a few sample runs.

Exercise W29.8

Update the shell script in Exercise W29.7 so that, after accomplishing this task, it sets the positional arguments to the output of the `who | head -1` command and displays the positional arguments again.

W29.6 COMMENTS AND PROGRAM HEADERS

You should develop the habit of putting comments in your programs to describe the purpose of a particular series of commands. At times, you should even briefly describe the purpose of a variable or assignment statement. Also, you should use a program header for every shell script that you write. These are simply good software engineering practices. A *program header* is a set of introductory comments used to explain the script. Program header and in-code comments help a programmer who has been assigned the task of maintaining (i.e., modifying or enhancing) your code to understand it quickly. They also help you understand your own code, in particular, if you reread it after some period of time. Long ago, putting comments in the program code or creating separate documentation for programs was not a common practice. Such programs when inherited by a programmer or a team are very difficult to understand and maintain, and are commonly known as *legacy code*. You may find different definitions for legacy code in the literature.

A good program header must contain at least the following items. In addition, you can insert any other items that you feel are important or are commonly used in your organization or group as part of its coding rules.

1. Name of the file containing the script
2. Name of the author
3. Date written
4. Date last modified
5. Purpose of the script (in one or two lines)
6. A brief description of the algorithm used to implement the solution to the problem at hand

A comment line, including every line in the program header, must start with the number sign (#), as in:

```
# This is a comment line.
```

However, a comment does not have to start at a new line; it can follow a command, as in:

```
set Var1=a Var2 Var3=b  # Assign "a" to Var1, "b" to Var3, and declare
                        # a variable Var2 with an initial value of null.
```

The following is a sample header for the **set_demo** script:

```
# File Name:           ~/linux2e/TCshell/set_demo
# Author:              Syed Mansoor Sarwar
# Date Written:        August 10, 1999 (by the original author)
# Last Modified:        January 22, 2017 (by the original author)
# Purpose:              To illustrate how the set command works
# Brief Description:    The script runs with a filename as the only command line
#                       argument, saves the filename, runs the set command to assign the
#                       output of the ls -il command to positional arguments ($1-$9),
#                       and displays file name, its inode number, file permissions, and
#                       file size (in bytes).
```

We do not show the program headers for all of the sample scripts in this textbook for the sake of brevity.

W29.7 PROGRAM CONTROL FLOW COMMANDS

The program control flow commands/statements are used to determine the sequence in which statements in a shell script execute. The four basic types of statements for controlling the flow of a script are two-way branching, multiway branching, repetitive execution of a group of commands/statements, and transferring control to a particular statement via a *jump* or *goto* statement of some sort. The TC shell statement for two-way branching is the `if` statement, the statements for multiway branching are the `if` and `switch` statements, and the statements for repetitive execution of some code are the `foreach` and `while` statements. In addition, the TC shell has a `goto` statement that allows you to jump to any command in a program.

W29.7.1 The `if-then-else-endif` Statement

The most basic form of the `if` statement is used for one-way branching, but the statement can also be used for multiway branching. The following is a brief description of the statement. The words in monospace type are *keywords* and must be used as shown in the syntax. Everything in brackets is optional. All the command lists are designed to help you accomplish the task at hand.

SYNTAX

```
if (expression) then
   then-command-list
   [else if (expression) then
      then-command-list
   ...
   else
      else-command-list]
endif
```

Purpose: To implement two-way or multiway branching

Here, an **expression** is a list of commands. The execution of commands in **expression** returns a status of true (success) or false (failure). We discuss three versions of the **`if`** statement that together comprise the statement's complete syntax and semantics. The most basic use of the **`if`** statement is without any optional features and results in the following syntax for the statement, which is commonly used for two-way branching.

SYNTAX

```
if (expression) then
   then-commands
endif
```

Purpose: To implement two-way branching

If **expression** is true, the **then-commands** are executed; otherwise, the command after, **`endif`** is executed.

The semantics of the statement are shown in Figure 29.1.

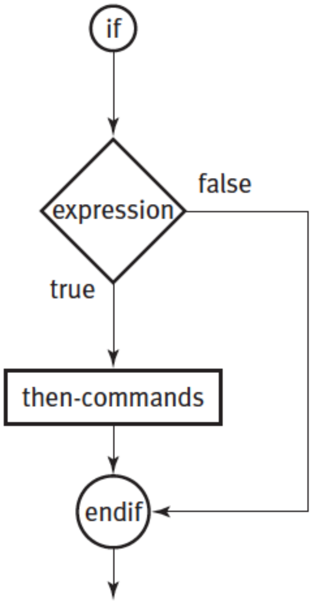

FIGURE W29.1 Semantics of the `if-then-if` statement.

You can form an expression by using a variety of operators for testing files, testing and comparing integers and strings, and logically connecting two or more expressions to form complex expressions. Table W29.4 describes the operators that can be used to form expressions, along with their meanings. Operators not related to files are listed in the order of their precedence (from high to low): parentheses, unary, arithmetic, shift, relational, bitwise, and logical.

TABLE W29.4 TC Shell Operators for Forming Expressions

Operator	Function	Operator	Function
Parentheses		**Relational**	
( )	To change the order of evaluation	>	Greater than
		<	Less than
		>=	Greater than or equal to
		<=	Less than or equal to
		!=	Not equal to (for string comparison)
		==	Equal to (for string comparison)
Unary		**Bitwise**	
–	Unary minus	&	AND
~	One’s complement	^	XOR (exclusive OR)
!	Logical negation	|	OR
Arithmetic		**Logical**	
%	Remainder	&&	AND
/	Divide	||	OR
*	Multiply		
–	Subtract		
+	Add		
Shift Operators			
>>	Shift right		
<<	Shift left		

File- and String-Related Operators

Operator	Function	Operator	Function	Operator	Function
-d file	True if **file** is a directory	**-e file**	True if **file** exists	**-f file**	True if **file** is ordinary file
-o file	True if user owns **file**	**-r file**	True if **file** is readable	**-w file**	True if **file** is writable
-x file	True if **file** is	**-z file**	True if length of **file** is		

executable	zero bytes

We use the preceding syntax of the `if` command to modify the script in the **set_demo** file so that it takes one command-line argument only and checks on whether the argument is a file or a directory. The script returns an error message if the script is run with none or more than one command-line argument, or if the command-line argument is not an ordinary file. The name of the script file is **if_demo1**. The contents of the file and its sample runs are as follows:

```
% cat if_demo1
#!/bin/tcsh
if ( ( $#argv == 0 ) || ( $#argv > 1 ) ) then
    echo "Usage: $0 ordinary_file"
    exit 1
endif
if ( ! -e $1 ) then
    echo "$1 : non-existent file"
    echo "Usage: $0 ordinary_file"
    exit 1
endif
if ( -d $1 ) then
    echo "$1 : directory"
    echo "Usage: $0 ordinary_file"
    exit 1
endif
if ( -f $1 ) then
    set filename = $argv[1]
    set fileinfo = `ls -il $filename`
    set inode = $fileinfo[1]
    set perms = $fileinfo[2]
    set size = $fileinfo[6]
    echo "File Name:        $filename"
    echo "Inode Number:     $inode"
    echo "Permissions:      $perms"
    echo "Size (bytes):     $size"
    exit 0
endif
echo "$0: argument must be an ordinary file"
exit 1
% ./if_demo1
Usage: if_demo1 ordinary_file
% ./if_demo1 lab1
lab1 : non-existent file
Usage: if_demo1 ordinary_file
% ./if_demo1 dir1
dir1 : directory
Usage: if_demo1 ordinary_file
% ./if_demo1 if_demo1
File Name:        if_demo1
Inode Number:     16517767
Permissions:      -rwxr-xr-x
Size (bytes):     699
% ls -il if_demo1
16517767 -rwxr-xr-x 1 sarwar faculty 699 Feb 19 18:18 if_demo1
%
```

In the preceding script, the first instance of the `if` statement that contains a compound expression displays an error message and exits the program if you run the script without a command-line argument or with more than one argument. The second and third `if` statements, respectively, check if the file specified as command-line argument exists and is not a directory. The fourth `if` statement is executed if you run the script with only one command-line argument that exists in your directory hierarchy and is not a directory. It produces the desired results if the command-line argument is an ordinary file. If the passed argument is not an ordinary file, the condition for the second `if` statement is false and the error message `if_demo1: argument must be an ordinary file` is displayed. Note that the exit status of the script is `1` when it exits because of an erroneous condition, and `0` when the script executes successfully and produces the desired results.

An important practice in script writing is to correctly indent the commands/statements in it. Proper indentation of programs enhances their readability, making them easier to understand, debug, and maintain (add or remove features). Note the indentation style used in the sample scripts presented in this textbook and follow it when you write scripts.

The second instance of the `if` statement syntax also allows two-way branching. The following is a brief description of the statement.

SYNTAX

`if` (expression) `then`

  then-commands

`else`

  else-commands

`endif`

Purpose: To implement two-way branching

If **expression** is true, the commands in **then-commands** are executed; otherwise, the commands in **else-commands** are executed, followed by the execution of the first command after **`endif`**. The semantics of the statement are shown in Figure W29.2.

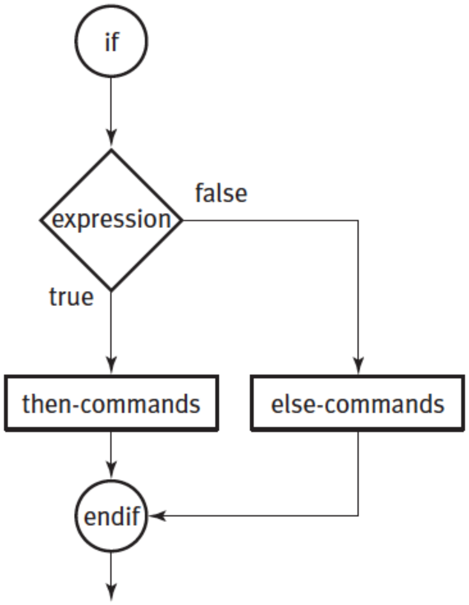

FIGURE W29.2 Semantics of the `if-then-else-endif` statement.

We rewrite the **if_demo1** program by using the `if-then-else-endif` statement at the end. The resulting script is in the **if_demo2** file, as shown in the following session. Notice that the program looks cleaner and more readable.

```
% cat if_demo2
#!/bin/tcsh
if ( ( $#argv == 0 ) || ( $#argv > 1 ) ) then
    echo "Usage: $0 ordinary_file"
    exit 1
endif
if ( ! -e $1 ) then
    echo "$1 : non-existent file"
    echo "Usage: $0 ordinary_file"
    exit 1
endif
if ( -d $1 ) then
    echo "$1 : directory"
    echo "Usage: $0 ordinary_file"
    exit 1
endif
if ( -f $1 ) then
    set filename = $argv[1]
    set fileinfo = `ls -il $filename`
    set inode = $fileinfo[1]
    set perms = $fileinfo[2]
    set size = $fileinfo[6]
    echo "File Name:        $filename"
    echo "Inode Number:     $inode"
    echo "Permissions:      $perms"
    echo "Size (bytes):     $size"
    exit 0
else
    echo "$0: argument must be an ordinary file"
    exit 1
endif
%
```

The third version of the `if` statement is used to implement multiway branching. The following is a brief description of the statement.

SYNTAX

```
if (expression1) then
   then-commands
else if (expression2) then
   else-if1-commands
else if (expression3) then
   else-if2-commands
...
else
   else-commands
endif
```

Purpose: To implement multiway branching

If **expression1** is true, the commands in **then-commands** are executed. If **expression1** is false, **expression2** is evaluated, and if it is true, the commands in **else-if1-commands** are executed. If **expression2** is false, **expression3** is evaluated. If **expression3** is true, **else-if2-commands** are executed. If **expression3** is also false, the commands in **else-commands** are executed. The execution of any command list is followed by the execution of the command after **`endif`**. You can use any number of `else-if`s in an `if` statement to implement multiway branching. The semantics of the statement are illustrated in Figure W29.3.

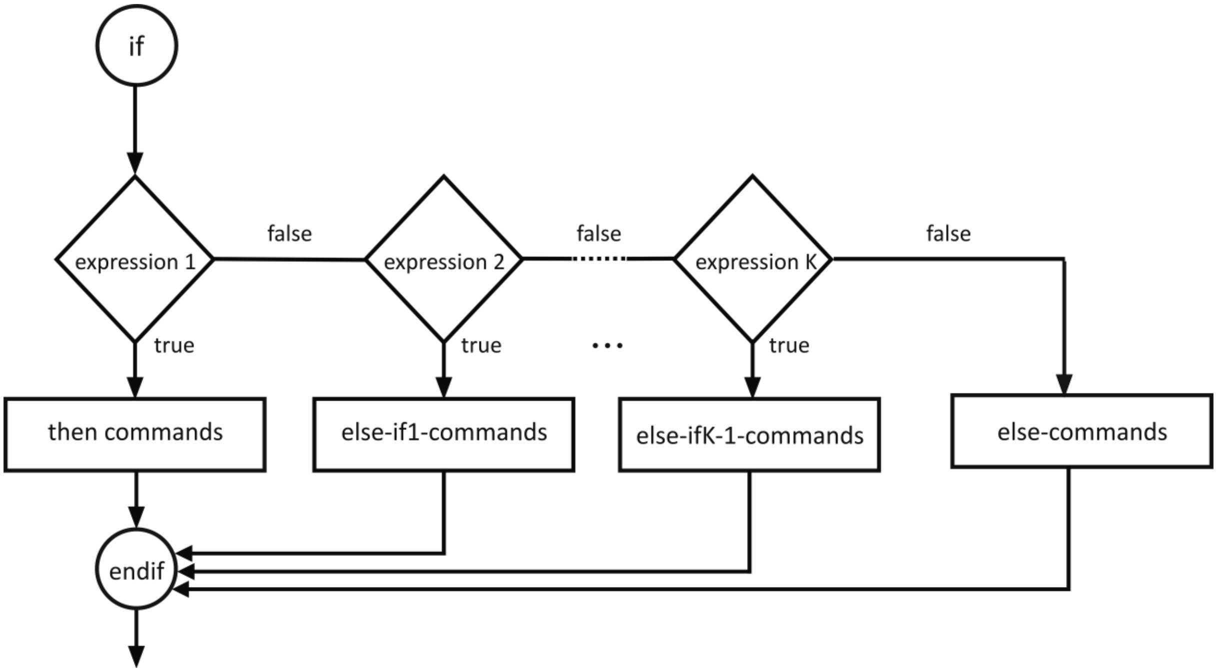

FIGURE W29.3 Semantics of the `if-then-else-if-else-endif` statement.

We enhance the script in the **if_demo2** script so that if the command-line argument is a directory, the program displays the number of files and subdirectories in it, excluding hidden files. If the directory is empty, the script informs you so. Implementation also involves the use of the `if-then-else-endif` statement throughout. The resulting script is in the **if_demo3** file, as shown in the following session. As you can see, although it contains more functionality, it is more elegant than the previous two versions.

```
% cat if_demo3
#!/bin/tcsh
if ( ( $#argv == 0 ) || ( $#argv > 1 ) ) then
    echo "Usage: $0 ordinary_file"
else if ( ! -e $1 ) then
    echo "$1 : non-existent file"
else if ( -d $1 ) then
    set nfiles = `ls $1 | wc -w`
```

```
        if ($nfiles == 0) then
            echo "$1 : Empty directory"
        else
            echo "The number of files in $1 is : $nfiles."
        endif
else if ( -f $1 ) then
    set filename = $argv[1]
    set fileinfo = `ls -il $filename`
    set inode = $fileinfo[1]
    set perms = $fileinfo[2]
    set size = $fileinfo[6]
    echo "File Name:        $filename"
    echo "Inode Number:     $inode"
    echo "Permissions:      $perms"
    echo "Size (bytes):     $size"
else
    echo "$0: argument must be an ordinary file"
endif
% ./if_demo3
Usage: if_demo3 ordinary_file
% ./if_demo3 lab1
lab1 : non-existent file
% ./if_demo3 dir1
dir1 : Empty directory
% ./if_demo3 .
The number of files in . is : 17.
% ./if_demo3 if_demo3
File Name:        if_demo3
Inode Number:     16517769
Permissions:      -rwxr-xr-x
```

```
Size (bytes):     747
% ls -il if_demo3
16517769 -rwxr-xr-x 1 sarwar faculty 747 Feb 19 18:22 if_demo3
%
```

If the command-line argument is an existing file, the required file-related data is displayed. If the argument is a directory, the number of files in it (including directories and hidden files) is saved in the `nfiles` variables and displayed. If the argument is a nonexistent file or directory, the error message `if_demo3: argument must be an existing file or directory` is displayed. The sample runs of the script show these cases. The same runs also show the expected outputs when the script is run without an argument and more than one argument.

In the following in-chapter exercises, you will practice the use of `if` statement, command substitution, and manipulation of positional parameters.

Exercise W29.9

Create the **if_demo2** script file and run it with no argument, more than one argument, and one argument only. While running the script with one argument, use a directory as the argument. What happens? Does the output of the script make sense?

Exercise W29.10

Write a shell script whose single command-line argument is a file. If you run the program with an ordinary file, the program displays the owner’s name and last update time for the file. If the program is run with more than one argument, it generates meaningful error messages.

W29.7.2 The `foreach` Statement

The `foreach` statement is the first of two statements available in the TC shell for repetitive execution of a block of commands in a shell script. These statements are commonly known as *loops*. The following is a brief description of the statement.

SYNTAX

```
foreach variable (argument-list)
   command-list
end
```

Purpose: To execute commands in **command-list** as many times as the number of strings in **command-list**; if **argument-list** is **$argv**, the arguments are taken from the command-line arguments

The strings in **argument-list** are assigned to **variable** one by one, and the commands in **command-list**, also known as the body of the loop, are executed for every assignment. This process allows execution of the commands in **command-list** as many times as the number of words in **argument-list**. Figure W29.4 depicts the semantics of the `foreach` command.

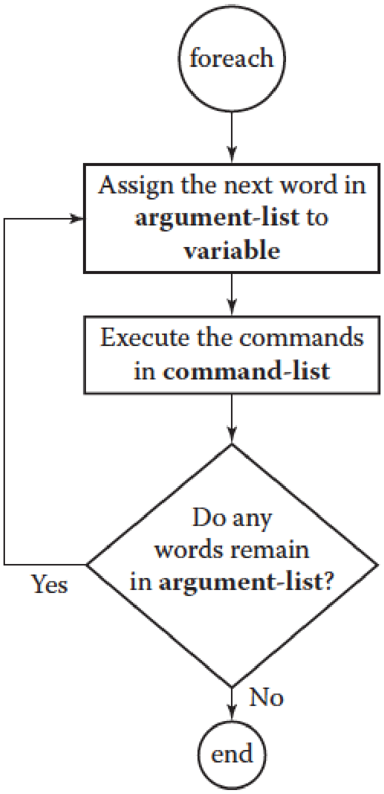
foreach
Assign the next word in argument-list to variable
Execute the commands in command-list
Do any words remain in argument-list?
Yes
No
end

FIGURE W29.4 Semantics of the `foreach` statement.

The following script in the **foreach_demo** file shows use of the `foreach` command with optional arguments. The variable `person` is assigned the strings in **argument-list** one by one each time the value of the variable is echoed, until no strings remain in the list. At this time, control comes out of the `foreach` statement, and the command following `end` is executed. Then, the code following the `foreach` statement (the `exit` statement only in this case) is executed.

```
% cat foreach_demo
#!/bin/tcsh
    foreach person ( Debbie Jamie John Kitty Kuhn Shah )
    echo $person
end
exit 0
% ./foreach_demo
Debbie
Jamie
John
Kitty
Kuhn
Shah
%
```

The following script in the **user_info** file takes a list of existing (valid) login names as command-line arguments and displays each login name and the full name of the user who owns the login name, one per line. In the sample run, the first value of the user variable is **dheckman**. The `echo` command displays `dheckman:` followed by a `<Tab>`, and the cursor stays at the current line. The `grep` command searches the **/etc/passwd** file for `dheckman` and pipes it to the `cut` command, which displays the fifth field in the **/etc/passwd** line for **dheckman** (his full name). The process is repeated for the remaining two login names (**ghacker** and **sarwar**). As no user is left in the list passed at the command line, control comes out of the `foreach` statement and the `exit 0` command is executed to transfer control back to shell. The command substitution ``"^"`echo $user":"`` in the `grep` command can be replaced by `"^"$user":"`. The `>&` operator is used to redirect the output and error message generated by the `grep` command to **/dev/null**, the Linux black hole. It is so done because the purpose of the `grep` command is to check whether a login name specified at the command line exists in **/etc/passwd** file or not. The subsequent `if` statement checks the return status (saved in `$?`) of the `grep` command; `0` means success.

```
% cat user_info
```

```
#!/bin/tcsh
foreach user ( $argv )
# Don't display anything if a login name is not found in /etc/passwd
   grep "^"`echo $user":"` /etc/passwd >& /dev/null
   if ( $? == 0 ) then
           echo -n "$user :       "
           grep "^"$user":" /etc/passwd | cut -f5 -d':'
   endif
end
exit 0
% ./user_info dheckman ghacker sarwar
rizwan :       Dennis R. Heckman
zaheer :       George Hacker
sarwar :       Syed Mansoor Sarwar
%
```

W29.7.3 The `while` Statement

The `while` statement, also known as the `while` loop, allows repeated execution of a block of code based on the condition of an expression. The following is a brief description of the statement. Figure W29.5 illustrates the semantics of the `while` statement.

SYNTAX

```
while (expression)
   command-list
end
```

Purpose: To execute commands in **command-list** so long as **expression** evaluates to true

The `expression` is evaluated and, if the result of this evaluation is true, the commands in `command-list` are executed and `expression` is evaluated again. This sequence of expression evaluation and execution of `command-list`, known as *iteration*, is repeated until the `expression` evaluates to false. At that time, control comes out of the `while` statement and the statement following `end` is executed.

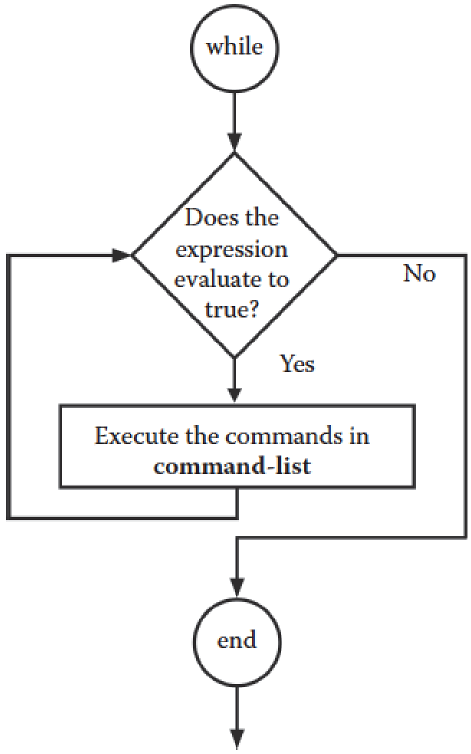
while
Does the expression evaluate to true?
No
Yes
Execute the commands in command-list
end

FIGURE W29.5 Semantics of the `while` statement.

The variables and/or conditions in the expression that result in a true value must be properly manipulated in the commands in `command-list` for well-behaved loops—that is, loops that eventually terminate and allow execution of the rest of the code in a script. Loops in which the expression always evaluates to true are known as *infinite loops*. Infinite loops, also known as *nonterminating loops*, are usually a result of poor design and/or programming, are undesirable because they continuously use CPU time without accomplishing any useful task. However, some applications require infinite loops. For example, all the servers for Internet services, such as the HTTP server, are programs that run indefinitely, waiting for client requests. Once a server has received a client request, it processes it, sends a response to the client, and waits for another client request. The only way to terminate a process with an infinite loop is to kill it by using the `kill` command. Or, if the process is running in the foreground, pressing `<Ctrl+C>` would do the trick, unless the process is designed to ignore `<Ctrl+C>`. In that case, you need to put the process in the background by pressing `<Ctrl+Z>` and using the `kill -9` command to terminate it. We discussed Linux processes, including the foreground and background processes, in Chapter 10.

The following script in the **while_demo** file shows a simple use of the `while` loop. When you run this script, the `secretcode` variable is initialized to `agent007`, and you are prompted to make a guess. Your guess is stored in the local variable `yourguess`. If your guess is not `agent007`, the condition for the `while` loop is true and the commands between `while` and `end` are executed. The program displays a tactful message informing you of your failure and prompts you for another guess. Your guess is again stored in the `yourguess` variable and the condition for the loop is tested. This process continues until you enter `agent007` as your guess. This time, the condition for the loop becomes false, and control comes out of the `while` statement. The `echo` command following `done` executes, congratulating you for being part of a great gene pool!

```
% cat while_demo
#!/bin/tcsh
set secretcode = agent007
echo "Guess the code!"
echo -n "Enter your guess: "
set yourguess = `head -1`
while ("$secretcode" != "$yourguess")
    echo Good guess but wrong. Try again!
    echo -n "Enter your guess: "
    set yourguess = `head -1`
end
echo "Wow! You are a genius\!!"
exit 0
% ./while_demo
Guess the code!
Enter your guess: star wars
Good guess but wrong. Try again!
Enter your guess: columbo
Good guess but wrong. Try again!
Enter your guess: agent007
Wow! You are a genius!!
```

```
%
```

W29.7.4 The `break`, `continue`, and `goto` Commands

The `break` and `continue` commands can be used to interrupt the sequential execution of the loop body. The `break` command transfers control to the command following `end`, thus terminating the loop prematurely. A good programming use of the `break` command is to transfer control out of a nested loop. The `continue` command transfers control to `end`, which results in the evaluation of the loop condition again, hence continuation of the loop. In both cases, the commands in the loop body following these statements are not executed. Thus, they are typically part of a conditional statement, such as an `if` statement. The `goto` command can be used to transfer control to any location in the script. The following is a brief description of the command.

SYNTAX
`goto` label
Purpose: To execute the command at the **label**

The `goto` command transfers control to the command at `label:`, a tag for the command. The use of `goto` is considered a bad programming practice because it makes debugging of programs a daunting task. For this reason, we do not recommend its use, with the exception perhaps of transferring control out of a nested loop—all loops and not just the one that has the `goto` command in it. Figure W29.6 illustrates the semantics of these commands.

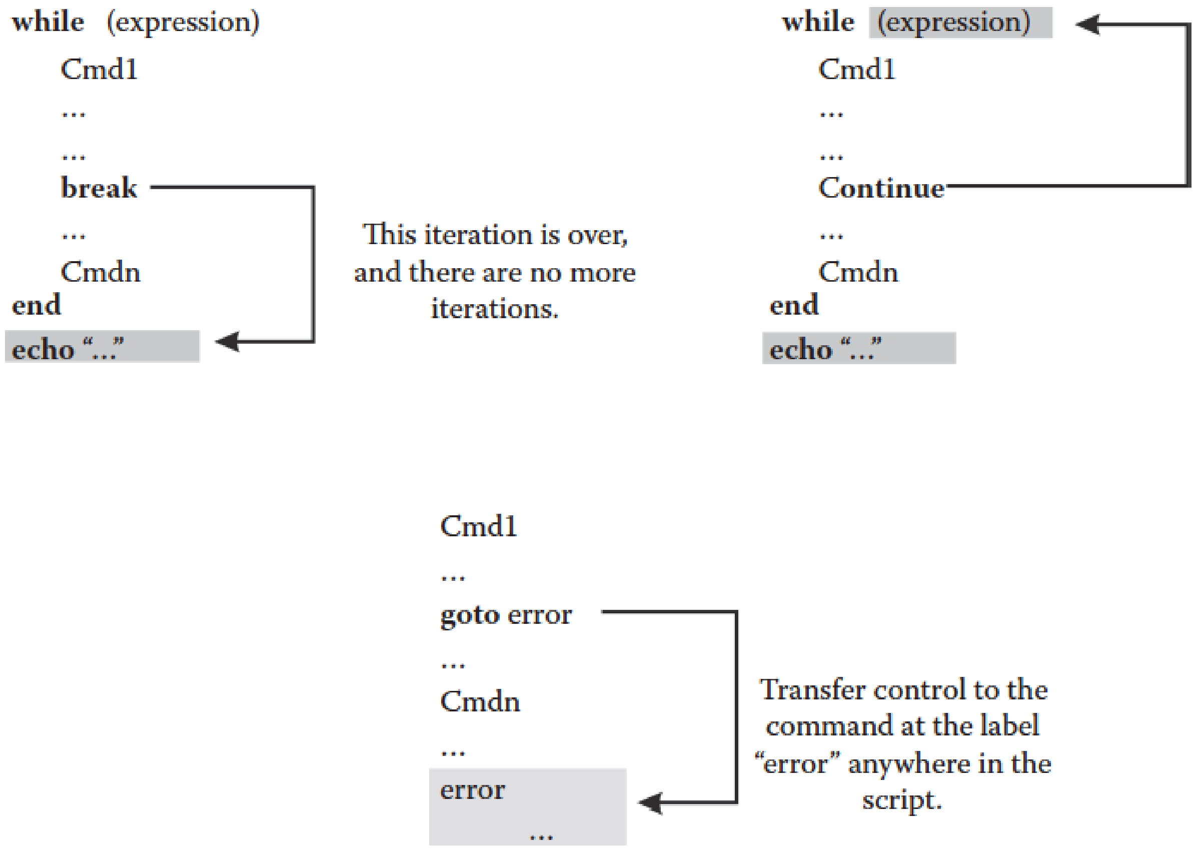
while (expression)
Cmd1
…
…
break
…
Cmdn
end
echo "…"
This iteration is over, and there are no more iterations.
while (expression)
Cmd1
…
…
Continue
…
Cmdn
end
echo "…"
Cmd1
…
goto error
…
Cmdn
…
error
…
Transfer control to the command at the label "error" anywhere in the script.

FIGURE W29.6 Semantics of the `break`, `continue`, and `goto` commands.

In the following in-chapter exercises, you will write the TC shell scripts with loops by using the `foreach` and `while` statements.

Exercise W29.11

Write a shell script that takes a list of host names on your network as command-line arguments and displays whether the hosts are up or down. Use the `ping` command to display the status of a host and the `foreach` statement to process all host names.

Exercise W29.12

Rewrite the script in Exercise W29.11 by using the `while` statement.

W29.7.5 The `switch` Statement

The `switch` statement provides a mechanism for multiway branching similar to a nested `if` statement. However, the structure provided by the `switch` statement is more readable. You should use the `switch` statement when you can—that is, when you are testing a single variable to several distinct patterns. You would not use it when you want to test more than one variable. The following is a brief description of the statement.

SYNTAX

```
switch (test-string)
   case pattern1:
      command-list1
      breaksw
   case pattern2:
      command-list2
      breaksw
   ...
   ...
   default:
      command-listN
      breaksw
endsw
```

Purpose: To implement multiway branching as with a nested `if`

The `switch` statement compares the value in **test-string** with the values of all the patterns one by one until either a match is found or there are no more patterns to match **test-string** with. If a match is found, the commands in the corresponding **command-list** are executed and control goes out of the `switch` statement. If no match is found, control goes to commands in **command-listN**. You don't need to include a default for the `switch` statement. Figure W29.7 illustrates the semantics of the `switch` statement.

The following script in the **switch_demo** file shows a simple but representative use of the `switch` statement. It is a menu-driven program that displays a menu of options and prompts you to enter an option. Your input is read into a variable called `option`. The `switch` statement then matches your option with one of the four available patterns (single characters in this case) one by one, and when a match is found, the corresponding **command-list** is executed. Thus, if you type `d` and hit `<Enter>` at the prompt, the `date` command is executed and control goes out of `switch`. The `exit 0` command is then executed for normal program termination. Note that the TC shell performs logical OR on the items enclosed in brackets. Thus, here, uppercase and lowercase letters are treated the same. A few sample runs of the script follow the code.

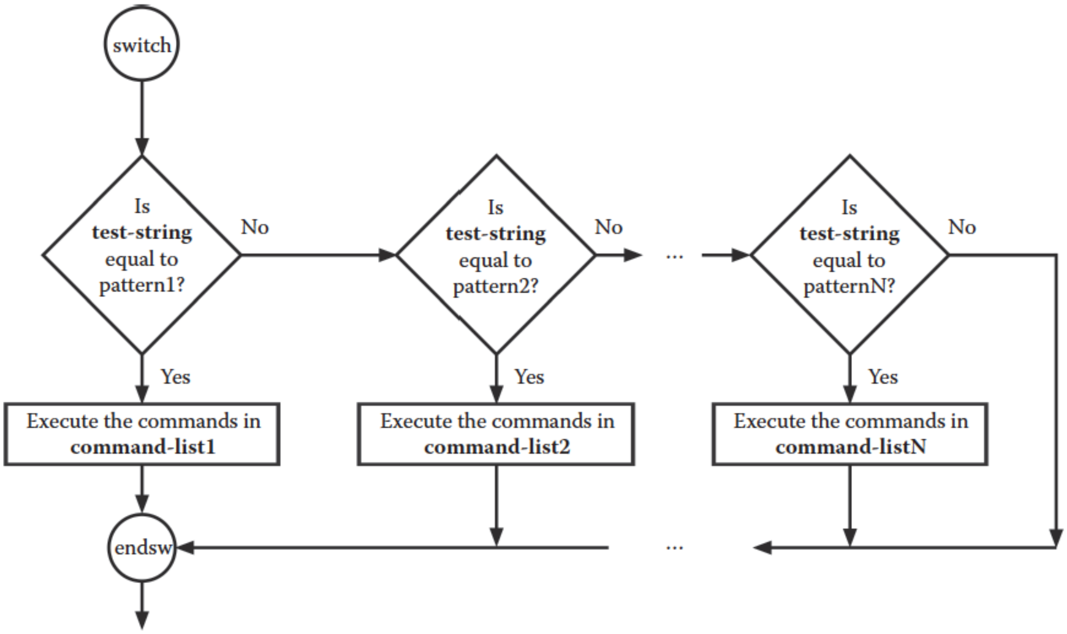
switch
Is test-string equal to pattern1?
No
Yes
Execute the commands in command-list1
Is test-string equal to pattern2?
No
Yes
Execute the commands in command-list2
...
Is test-string equal to patternN?
No
Yes
Execute the commands in command-listN
endsw
...

FIGURE W29.7 Semantics of the `switch` statement.

```
% cat switch_demo
#!/bin/tcsh
echo "Use one of the following options:"
echo " d or D: To display today’s date and present time"
echo " l or L: To see the listing of files in your present working directory"
echo " w or W: To see who is logged in"
echo " q or Q: To quit this program"
echo -n "Enter your option and hit <Enter>: "
set option = `head -1`
switch ("$option")
    case [dD]:
        date
        breaksw
    case [lL]:
        ls
        breaksw
    case [wW]:
        who
        breaksw
    case [qQ]:
        exit 0
        breaksw
    default:
        echo "Invalid option; try running the program again."
        exit 1
        breaksw
endsw
exit 0
% ./switch_demo
Use one of the following options:
 d or D: To display today’s date and present time
 l or L: To see the listing of files in your present working directory
 w or W: To see who is logged in
 q or Q: To quit this program
Enter your option and hit <Enter>: d
Sun Feb 19 18:33:24 PKT 2017
% ./switch_demo
Use one of the following options:
 d or D: To display today’s date and present time
 l or L: To see the listing of files in your present working directory
 w or W: To see who is logged in
 q or Q: To quit this program
Enter your option and hit <Enter>: L
cmdargs_demo         display_name  if_demo1  keyin_demo  switch_demo
```

```
dir1                 export_demo   if_demo2  set_demo    user_info
display_change_name  foreach_demo  if_demo3  shift_demo  while_demo
% ./switch_demo
Use one of the following options:
 d or D: To display today’s date and present time
 l or L: To see the listing of files in your present working directory
 w or W: To see who is logged in
 q or Q: To quit this program
Enter your option and hit <Enter>: q
% ./switch_demo
Use one of the following options:
 d or D: To display today’s date and present time
 l or L: To see the listing of files in your present working directory
 w or W: To see who is logged in
 q or Q: To quit this program
Enter your option and hit <Enter>: a
Invalid option; try running the program again.
%
```

SUMMARY

Every Linux shell has a programming language that allows you to write programs for performing tasks that cannot be performed by using the existing commands. These programs are commonly known as shell scripts. In its simplest form, a shell script consists of a list of shell commands that are executed by a shell sequentially, one by one. More advanced scripts use program control flow statements for implementing multiway branching, repetitive execution of a block of commands in the script, transferring control out of a loop, or transferring control to any command in the script. The shell programs that consist of TC shell commands, statements, and features are called TC shell scripts.

The shell variables are the main memory locations that are given names and can be read from and written to by using these names, instead of the addresses of the relevant memory locations. There are two types of shell variables: environment variables and user-defined variables. Environment variables, initialized by the shell at the time the user logs on, are maintained by the shell to provide a user-friendly work environment. User-defined variables are used as scratch pads in a script to accomplish the task at hand. Some environment variables, such as the positional parameters, are read only in the sense that the user cannot change their values without using the `set` command.

The TC shell commands for processing shell variables are `set` and `setenv` (for setting values of positional parameters and displaying values of all environment variables), `env` (for displaying values of all shell variables), `unset` and `unsetenv` (for removing shell variables from the environment set by using the `set` and `setenv` commands, respectively), `set` with `$<`, `"$<"`, `` `echo $<` ``, or `` `head -1` `` (for assigning keyboard input as values of variables), and `shift` (for shifting command-line arguments to the left by one or more positions).

The program control flow statements `if` and `switch` allow the user to implement multiway branching, and the `foreach` and `while` statements can be used to implement loops. The `continue`, `break`, and `goto` commands can be used to interrupt the sequential execution of a shell program and transfer control to a statement that (usually) is not the next statement in the program layout.

Questions and Problems

Note: All scripts should be written using the TC shell language.

1. What is a shell script? Describe three ways of executing a shell script.
2. What is a shell variable? What is a read-only variable? How can you make a user-defined variable read only? Give an example to illustrate your answer.

3. Which shell environment variable is used to store your search path? Change your search path interactively to include the directories **~/bin** and **.** . What would this change allow you to do? Why? If you want to make it a permanent change, what would you do? See Chapter 2 if you have forgotten how to change your search path.

4. The `echo "\044"` command displays $ on the screen. Why?

5. What will be the output if the shell script **keyin_demo** in Section W29.3.5 is executed and you give `*` as input each time you are prompted?

6. Write a shell script that takes an ordinary file as an argument and removes the file if its size is zero. Otherwise, the script displays the file's name, size, number of hard links, owner, and modify date (in this order) on one line. Your script must do the appropriate error checking.

7. Write a shell script that takes a directory as a required argument and displays the name of all zero-length files in it. Do the appropriate error checking.

8. Write a shell script that removes all zero-length ordinary files and empty directories in the current directory. Do the appropriate error checking.

9. Modify the script in Problem 7 so that it removes all zero-length ordinary files in the directory passed as an optional argument. If you do not specify the directory argument, the script uses the present working directory as the default argument. Do the appropriate error checking.

10. Run the script in **if_demo2** in Section W29.6 with `if_demo2` as its argument. Does the output make sense to you? Why?

11. Modify the script in **if_demo2** in Section W29.6 so that it takes two ordinary files as arguments and displays information about both. Also, the script should display an error message if a special file is passed to it as an argument.

12. Write a shell script that takes a list of login names on your computer system as command-line arguments and displays these login names and full names of the users who own these logins

(as contained in the **/etc/passwd** file), one per line. If a login name is invalid (not found in the **/etc/passwd** file), display the login name but nothing for the full name. The format of the output line is `login name: user name`.

13. What happens when you run a stand-alone command enclosed in back quotes (grave accents), such as `` `date` ``? Why?

14. What happens when you execute the following sequence of shell commands?

    a. `set name=date`

    b. `$name`

    c. `` `$name` ``

15. Take a look at your **~/.login** and **~/.cshrc** files and list the environment variables that are exported, along with their values. What is the purpose of each variable?

16. What are the following commands used for? What is the difference between these commands?

    a. `set var1 = $<`

    b. `set var1 = "$<"`

    c. `` set var1 = `echo $<` ``

    d. `` set var1 = `head -1` ``

17. Write a shell script that takes a list of login names as its arguments and displays the number of terminals that each user is logged on to in a LAN environment.

18. Write a shell script `domain2ip` that takes a list of domain names as command-line arguments and displays their IP addresses. Use the `nslookup` command. The following is a sample run of this program.

```
% domain2ip up.edu linuxmint.com
Name: up.edu
Address: 64.251.254.23
```

```
Name: linuxmint.com

Address: 192.124.249.9

%
```

19. Modify the script in the **switch_demo** file in Section W29.6.5 so that it allows you to try any number of options and quits only when you use the `q` option.

20. Rewrite the `if_demo3` program in Section W29.7.1 using the `switch` statement.

21. Write a shell script that displays the following menu and prompts for one-character input to invoke a menu option, as follows:

    a. List all files in the present working directory

    b. Display today's date and time

    c. Invoke the shell script for Problem W15

    d. Display whether a file is a *simple* file or a *directory*

    e. Create a backup for a file

    f. Start a telnet session

    g. Start an ftp session

    h. Exit

    Option (c) requires that you ask the user for a list of login names. For options (d) and (e), prompt the user for file names before invoking a shell command/program. For options (f) and (g), prompt the user for a domain name (or IP address) before initiating a telnet or ftp session. The program should allow the user to try any option any number of times and should quit only when the user gives option (h) as input. A good programming practice is to build code incrementally—that is, write code for one option, test it, and then go to the next option. Use this style of code development for writing this shell script.

22. Write a shell script that reads a string from the keyboard. If the string is a pathname for a directory in your system’s directory structure, the script displays the counts of all types of files in it in the following format:

```
Directories                :

Block special files        :

Character special files    :

Link files                 :

FIFOs                      :

Ordinary files             :

Sockets                    :
```

The script displays the cumulative size (in bytes) for all ordinary files in the directory. Do the appropriate exception handling.