CHAPTER W30

Advanced TC Shell Programming

Objectives

- To discuss numeric data processing
- To describe array processing
- To discuss how standard input of a command in a shell script can be redirected to data within the script
- To explain signal/interrupt processing capability of the TC shell
- To cover the commands and primitives
  `=, +=, -=, *=, /=, %=, ==, <, >, |, &, (), <<, @, <Ctrl+Z>, <Ctrl+C>, clear, kill, onintr, set, stty`

W30.1 INTRODUCTION

We did not discuss four features of TC shell programming in Chapter W29: *processing of numeric data*, *array processing*, the *here document*, and *interrupt processing*. In this chapter, we discuss these features and give some example scripts that use them. We also describe how TC shell scripts can be debugged.

W30.2 NUMERIC DATA PROCESSING

The TC shell has a built-in capability for processing numeric data. It allows you to perform arithmetic and logic operations on numeric integer data without explicitly converting string data to numeric data and vice versa. You can use the `@` command to declare numeric variables, the variables that contain integer data. This command allows declaration of local variables only. The following is a brief description of the command.

SYNTAX
`@ variable [operator expression] [variable [operator expression]] ...`
Purpose: To declare `variable` to be a numeric variable, evaluate the arithmetic expression, apply the operator specified in `operator` to the current value of `variable` and the value of `expression`, and assign the result to `variable`

Expressions are formed by using the arithmetic and logic operators summarized previously in Table W29.4. Although octal numbers can be used in expressions by starting them with 0, the final value of an arithmetic expression is always expressed in decimal numbers. The elements of an expression must be separated by one or more spaces unless the elements are `(,)`, `&`, `|`, `<`, and `>`. Table W30.1 describes the possible assignment operators that can be used.

In the following interactive session, the numeric variables `value1` and `value2` are initialized to 10 and 15,

respectively. Note that no space is required before or after the = sign. The `echo` command is used to show that the assignments work properly.

```
% @ value1 = 10
% @ value2 = 15
% echo "$value1 $value2"
10 15
%
```

In the following session, the `@` command declares two variables, `difference` and `sum`, and initializes them to the values of the expressions to the right of the respective = signs. These actions result in the variables `difference` and `sum` taking the values −5 and 25, respectively.

```
% @ difference = ( $value1 - $value2 ) sum = ( $value1 + $value2 )
% echo $difference $sum
-5 25
%
```

You can use the `++` and `--` operators, also used in most contemporary high-level languages including C, C++, and Java, to increment or decrement a variable's value by 1. Thus, `@ var1++` and `@ var1--` increase and decrease, respectively, the value of the numeric variable `var1`. Similarly, we can use the following syntax to increment or decrement the value of `var1` by `N`.

```
@ var1 += N
@ var1 -= N
```

In the following session, we show with examples how you can use these operators. We assume that variables `value1` and `value2` have been initialized to 10 and 15, respectively, as shown in the earlier session.

```
% @ result = $value1 + $value2
% echo $result
25
% @ result++
% echo $result
26
% @ result += 1
% echo $result
27
% @ result = ( $result + 1 )
% echo $result
28
% @ result--
% echo $result
27
% @ result -= 1
% echo $result
26
% @ result -= $value1
% echo $result
16
%
```

You can also use the variables declared the `set` command to store numeric data. Thus, in the following session, the variables `side`, `area`, and `volume` are declared by using the `set` command and are assigned numeric values by using the `@` command. The `echo` command displays the values of these variables.

```
% set side = 10 area volume
% @ area = $side * $side
% @ volume = $side * $side * $side
% echo $side $area $volume
10 100 1000
%
```

In the following in-chapter exercise, you will perform numeric data processing by using the `set` and `@` commands.

Exercise W30.1

Declare two numeric variables `var1` and `var2` initialized to 10 and 30, respectively. Give two versions of a command that will produce and display their sum and product.

W30.2.1 ARITHMETIC WITH bc

bc is a language that supports arithmetic operations on arbitrary-precision numbers. It can be used as an interactive mathematical calculator or as a mathematical scripting language with C language like syntax. It supports the usual constructs and features of a scripting language to support arithmetic, assignment, comparison, increment/decrement, and logical/Boolean operations. It also supports conditional and iterative statements, and mathematical functions available in the Linux mathematics library, mathlib. You can also convert numbers from any number base to another, such as decimal to binary, binary to octal, octal to hexadecimal, and a number in base four to a number in base 21. We dicsus bc in detail in Chapter 13, Section 13.2.2.

W30.3 ARRAY PROCESSING

An array is a named collection of items of the same type stored in contiguous memory locations. Array items are numbered according to their locations in the array, with the first item being at location 1. You can access an array item by using the name of the array followed by the item number in brackets. Thus, you can use `people[k]` to refer to the *k*th element in the array called `people`. This process is known as array indexing. You can declare arrays for strings and integers by using the `set` command in the following manner.

SYNTAX
set array_name = (array elements)
Purpose: To declare `array_name` to be an array variable containing “array elements” in parentheses

You can access the contents of the whole array by using the array name preceded by the dollar sign `($)`, such as `$name`. You can access the total number of elements in an array by using the array name preceded by

$#, as in $#name, or simply by using $#, as is the case in the Bourne shell. The value of $?name is 1 if the named array has been initialized and 0 if it has not been initialized.

In the following session, we define a string array of six items, called students, initialized to the strings enclosed in parentheses. The contents of the whole array can be accessed by using $students, as shown in the first echo command. Thus, the echo $#students command displays 6 (the size of the students array), and the echo $?students command displays 1, informing you that the array has been initialized. We also show how you can declare an empty array, called empty, and initialize an array with the elements of another array; in this case, the empty array.

```
% set students = (David James Jeremy Brian Art Charlie)
% echo $students
David James Jeremy Brian Art Charlie
% echo $#students
6
% echo $?students
1
% set empty = ()
% echo $empty

% echo $#empty
0
% echo $?empty
1
% set students = ( $empty )
% echo $students

%
```

In the following session, we show how elements of the students array can be accessed and changed. You can access the *i*th element in the students array by indexing it as $students[i]. You can display a range of array elements from element at position i to the element at position j by using i-j as the index value, as in $students[i-j]. In the session later, the set command changes the second element of the students array from "James" to "Steve Jobs." Figure W30.1 depicts the original and modified students arrays. The first echo command displays all the elements of the array. The second echo command displays the second item in the students array. The third echo command displays four elements of the students array, starting with the item at position 2. The second set command, where we try to assign a value to the seventh slot of the students array, shows that, once defined, the size of an array is fixed and may not be changed. However, you may redefine an array of a new size by assigning it a new set of values, as shown via the third set and fourth echo commands in the following session. Note that the newly defined students array contains 11 elements. Finally, you may assign a subarray of an array to another array variable, as shown in the final two set and echo commands.

```
% set students[2] = "Steve Jobs"
% echo $students
David Steve Jobs Jeremy Brian Art Charlie
% echo $students[2]
Steve Jobs
% echo $students[2-5]
Steve Jobs Jeremy Brian Art
% set students[7] = Jamal
set: Subscript out of range.
```

```
% set students = ( "Dennis Ritchie" "Ken Thompson" David James Jeremy "Linus Torvalds"
Brian Art Charlie Jamal "Steve Jobs" )
% echo $students
Dennis Ritchie Ken Thompson David James Jeremy Linus Torvalds Brian Art Charlie Jamal
Steve Jobs
% set Linux_Author = ( $students[6] )
% echo $Linux_Author
Linus Torvalds
% set UNIX_Author = ( $students[1-2] )
% echo $UNIX_Author
Dennis Ritchie Ken Thompson
%
```

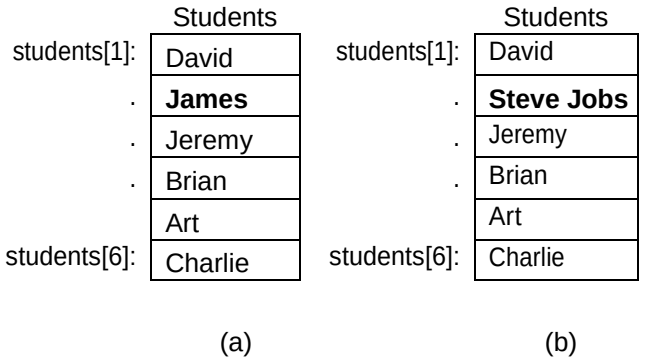

Figure W30.1 The *students* array (a) before and (b) after changing the contents of the second slot

Like other variables, an array variable can be removed from the environment by using the `unset` command. In the following session, the `unset` command is used to deallocate the `students` array. The `echo` command is used to confirm that the array has actually been deallocated.

```
% unset students
% echo $students
students: Undefined variable.
%
```

Any shell variable assigned multiple values with the `set` command becomes an array variable. Thus, when a variable is assigned a multiword output of a command as a value, it becomes an array variable and contains each field of the output, as separated by spaces, in a separate array slot. In the following session, `files` is an array variable whose elements are the names of all the files in the present working directory. The `numfiles` variable contains the number of files in the current directory. The `echo $files[3]` command displays the third array element.

```
% set files = `ls` numfiles = `ls | wc -w`
% echo $files
cmdargs_demo foreach_demo1 if_demo1 if_demo2 keyin_demo
% echo $numfiles
5
% echo $files[3]
```

```
if_demo1
%
```

You can also use an array declared with the `set` command to contain numeric data. In the following session, the **num_array_demo** file contains a script that uses an array of integers, called `Fibonacci`, computes the sum of integers in the array, and displays the sum on the screen. The `Fibonacci` array contains the first 10 numbers of the Fibonacci series. If you are not familiar with the Fibonacci series, the first two numbers in the series are 0 and 1 and the next Fibonacci number is calculated by adding the preceding two numbers. Therefore, the first 10 numbers in the Fibonacci series are 0, 1, 1, 2, 3, 5, 8, 13, 21, and 34. Thus, the Fibonacci series may be expressed mathematically as follows.

$$F_1 = 0, F_2 = 1, \ldots, F_n = F_{n-1} + F_{n-2} \quad (\text{for } n \geq 3)$$

The following script in the **num_array_demo** file is well documented and fairly easy to understand. It displays the sum of the first 10 Fibonacci numbers. A sample run of the script follows the code.

```
% cat num_array_demo
# File Name:       ~/linux2e/TCShell/num_array_demo
# Author:          Syed Mansoor Sarwar
# Written:         August 16, 2004
# Last Modified:   August 16, 2004; January 14, 2017
# Purpose:         To demonstrate working with numeric arrays
# Brief Description:Maintain running sum of numbers in a numeric variable called
#                  sum, starting with 0. Read the next array value and add
#                  it to sum. When all elements have been read, stop and
#                  display the answer.

#!/bin/tcsh

# Initialize the Fibonacci array to any number of Fibonacci numbers - first ten in this case

set Fibonacci = ( 0 1 1 2 3 5 8 13 21 34 )
@ size = $#Fibonacci      # Size of the Fibonacci array
@ index = 1               # Array index initialized to point to the first element
@ sum = 0                 # Running sum initialized to 0
while ( $index <= $size )
   @ sum = $sum + $Fibonacci[$index]    # Update the running sum
   @ index++                            # Increment array index by 1
end
echo "The sum of the given $#Fibonacci numbers is $sum."
exit 0
% ./num_array_demo
The sum of the given 10 numbers is 88.
%
```

Although this example clearly explains numeric array processing, it is of little or no practical use. We now present a more useful example, wherein the **fs** file contains a script that takes a directory as an optional argument and returns the size (in bytes) of all ordinary/plain files in it. A symbolic link that points to an ordinary file is considered an ordinary file. If no directory name is given at the command line, the script uses the current directory.

If you specify more than one argument, the script displays an error message and terminates. When executed with one nondirectory argument only, the program again displays an error message and exits. When executed with one argument only and the argument is a directory, the program initializes the string array variable `files` to the names of all the files in the specified directory by using the `set files = `ls $directory`` command. A numeric variable `nfiles` is initialized to the number of files in the specified directory by using the `@ nfiles = $#files` command. Then, the size of every ordinary file in the `files` array is added to the numeric variable `sum` that is initialized to `0`. When no more names are left in the `files` array, the program displays the total space (in bytes) used by all ordinary files in the directory, and then terminates.

```
% cat fs
#!/bin/tcsh

# File Name:       ~/linux2e/TCShell/fs
# Author:          Syed Mansoor Sarwar
# Written:         August 16, 2004
# Modified:        May 11, 2014; January 14, 2017
# Purpose:         To add the sizes of all the files in a directory passed as
#                  command line argument
# Brief Description: Maintain running sum of file sizes in a numeric variable
#                  called sum, starting with 0. Read all the file names
#                  by using the pipeline of ls, more, and while commands.
#                  Get the size of the next file and add it to the running
#                  sum. Stop when all file names have been processed and
#                  display the answer.

if ( $# == 0 ) then       # If no command line argument, the
                          # set directory to current directory
    set directory = "."
   else if ( $# != 1 ) then      # If more then one command line argument
                                 # then display command syntax
    echo "Usage: $0 [directory name]"
    exit 1
   else if ( ! -e "$1" ) then    # If one command line argument, but file
                                 # does not exist, display error message
        echo "$1 : File does not exist"
        exit 1
   else if ( ! -d "$1" ) then    # If one command line argument, but is
                                 # not a directory, show command syntax
    echo "Usage: $0 [directory name]"
    exit 1
   else
    set directory="$1"    # If one command line argument and it is a
                          # directory, prepare to perform the task
endif

# Initialize files array to file names in the specified directory
set files = `ls $directory`
@ nfiles = $#files # Number of files in the specified directory into nfiles
@ index = 1     # Array index initialized to point to the first file name
@ sum = 0       # Running sum initialized to 0

if ( "$nfiles" == 0 ) then
    echo "$directory : Empty directory"
    exit 0
endif
```

```
while ( $index <= $nfiles )      # For as long as a file name is left in files
    set thisfile = $directory/$files[$index]
    if ( -f "$thisfile" ) then       # If the next file is an ordinary file
    set argv = `ls -l $thisfile`     # Set command line arguments
    @ sum = $sum + $argv[5]          # Add file size to the running total
    @ index++
    else
    @ index++
    endif
end
echo "The size of all ordinary file(s) in $directory is $sum bytes."
exit 0
%
```

The following session shows a few sample runs of the **fs** script. Note that in the output of the `./fs /dev` command shows that the size of ordinary files in the **/dev** directory is 11 bytes. If you see the long listing of the **/dev** directory, you will notice that there is no ordinary file in it. However, the **/dev/core** file is a symbolic link to an ordinary file, **/proc/kcore**, and the `fs` command shows the size of the **/dev/core** file.

```
% ./fs linux2e
linux2e : File does not exist
% ./fs / /usr/bin
Usage: fs [directory name]
% ./fs dir1
d1 : Empty directory
% ./fs
The size of all ordinary files in . is 7360 bytes.
% ./fs ~
The size of all ordinary files in /home/sarwar is 0 bytes.
% ./fs ~/linux2e
The size of all ordinary files in /home/sarwar/linux2ee is 0 bytes.
% ./fs /
The size of all ordinary files in / is 61 bytes.
% ./fs /usr/bin
The size of all ordinary files in /usr/bin is 232019634 bytes.
% ./fs /dev
The size of all ordinary files in /dev is 11 bytes.
%
```

In the following in-chapter exercise, you will write a TC shell script that uses the numeric data processing commands for manipulating numeric arrays.

Exercise W30.2

Write a TC shell script that contains two numeric arrays, `array1` and `array2`, initialized to values in the sets {1, 2, 3, 4, 5} and {1, 4, 9, 16, 25}, respectively. The script produces and displays an array whose elements are the sum of the corresponding elements in the two arrays. Thus, the first element of the resultant array is 1 + 1 = 2, the second element is 2 + 4 = 6, and so on.

W30.4 THE HERE DOCUMENT

The *here document* feature of the TC shell allows you to redirect standard input of a command in a script and attach it to data within the script. Obviously, then, this feature works with commands that take input from standard input. The feature is used mainly to display menus, although it is also useful in other ways. The

following is a brief description of the here document.

SYNTAX
command <<[-] input_marker … **input data** … **input_marker**
Purpose: To execute `command` with its input coming from the here document—data between the input start and end markers "input_marker"

The "input_marker" is a word that you choose to wrap the input data in for `command`. The closing marker must be on a line by itself and cannot be surrounded by any spaces. The command and variable substitutions are performed before the here document data is directed to the standard input of the command. Quotes can be used to prevent these substitutions or to enclose any quotes in the here document. The "input_marker" can be enclosed in quotes to prevent any substitutions in the entire document, as follows:

```
command <<Marker
...
Marker
```

A hyphen (−) after << can be used to remove leading tabs (not spaces) from the lines in the here document and the marker that ends the here document. This feature allows the here document and the delimiting marker to conform to the indentation of the script. The following session illustrates this point:

```
while [ ... ]
do
    grep ... <<- DIRECTORY
          John Doe ...
          ...
          Art Pohm ...
    DIRECTORY
    ...
done
```

One last, but very important point: The output and error redirections of the command that uses the here document must be specified in the command line, not following the here document ending marker. The same is true of connecting the standard output of the command with other commands via a pipeline, as shown in the following session. Note that the `grep ... <<- DIRECTORY 2> errorfile | sort` command can be replaced by `(grep ... 2> errorfile | sort) <<- DIRECTORY`.

```
while [ ... ]
do
    grep ... <<- DIRECTORY 2> errorfile | sort
          John Doe ...
          ...
```

```
        Art Pohm ...
    DIRECTORY
    ...
done
```

We explain the use of the here document with a simple redirection of the **stdin** of the `cat` command from the document. The following script in the **heredoc_demo** file displays a message for the user and then sends a mail message to the person whose e-mail address is passed as a command line argument. Note that the Linux `mail` utility must have been installed before you can use this script.

```
% cat heredoc_demo
#!/bin/tcsh

cat << DataTag
This is a simple use of the here document. This data is the
input to the cat command.
DataTag

# Second example
mail -s "Weekly Meeting Reminder" $argv[1] << WRAPPER

Hello,

This is a reminder for the weekly faculty meeting tomorrow.

Mansoor

WRAPPER

echo "Sending mail to $argv[1] ... done."
exit 0
% ./heredoc_demo ecefaculty
This is a simple use of the here document. This data is the
input to the cat command.
Sending mail to ecefaculty ... done.
%
```

The following script is more useful and a better use of the here document feature. This script, `dext` (directory expert), maintains a directory of names, phone numbers, and e-mail addresses. The script is run with a name as a command-line argument and uses the `grep` command to display the directory entry corresponding to the name. The `-i` option is used with the `grep` command to ignore the case of letters.

```
% more dext
#!/bin/tcsh
if ( $#argv == 0 ) then
    echo "Usage: $0 name"
    exit 1
endif
set user_input = "$argv[1]"
grep -i "$user_input" << DIRECTORY

        John Doe        555.232.0000    johnd@somedomain.com
        Jenny Great     444.6565.1111   jg@new.somecollege.edu
        David Nice      999.111.3333    david_nice@xyz.org
```

```
        Don Carr        555.111.3333    dcarr@old.hoggie.edu
        Masood Shah     666.010.9820    shah@Garments.com.pk
        Jim Davis       777.000.9999    davis@great.adviser.edu
        Art Pohm        333.000.8888    art.pohm@great.professor.edu
        David Carr      777.999.2222    dcarr@net.net.gov

DIRECTORY
exit 0
% ./dext
Usage: ./dext name
% ./dext Pohm
        Art Pohm        333.000.8888    art.pohm@great.professor.edu
% ./dext Carr
        Don Carr        555.111.3333    dcarr@old.hoggie.edu
        David Carr      777.999.2222    dcarr@net.net.gov
% ./dext david
        David Nice      999.111.3333    david_nice@xyz.org
        David Carr      777.999.2222    dcarr@net.net.gov
%
```

The advantage of maintaining the directory within the script is that it eliminates some extra file operations such as open and read that would be required if the directory data were maintained in a separate file. The result is a much faster program.

Completing the following in-chapter exercise will enhance your understanding of the here document feature of TC shell.

Exercise W30.3

Create the `dext` script on your system and run it. Try it with as many different inputs as you can think of. Does the script work correctly?

W30.5 INTERRUPT (SIGNAL) PROCESSING

We discussed the basic concept of signals in Chapter 10, where we defined them as software interrupts that can be sent to a process. We also stated that the process receiving a signal can take any one of three possible actions:

1. Accept the default action as determined by the Linux kernel
2. Ignore the signal
3. Take a programmer-defined action

In Linux, several types of signals can be sent to a process. Some of these signals can be sent via the hardware devices such as the keyboard, but all can be sent via the `kill` command, as discussed in Chapters 10 and 13. The most common event that causes a hardware interrupt (and a signal) is generated when you press `<Ctrl+C>` and is known as the keyboard interrupt. This event, as its default action, causes the foreground process to terminate. Other events that cause a process to receive a signal include termination of a child process, a process accessing a main memory location that is not part of its address space (the main memory area that the program owns and is allowed to access), and a software termination signal caused by execution of the `kill` command without any signal number. Table W30.2 presents a list of some important signals, their numbers (which can be used to generate those signals with the `kill` command), and their purposes.

The signal processing feature of the TC shell allows you to write programs that cannot be terminated by a

terminal interrupt (<Ctrl+C>). In contrast to the Bourne shell support for signal processing, this feature is very limited. The command used to intercept and ignore <Ctrl+C> is onintr. The following is a brief description of the command.

SYNTAX	
onintr [options]	
Purpose:	To ignore a terminal interrupt (<Ctrl+C>) or intercept it and transfer control to any command
Commonly used options/features:	
–	To ignore the terminal interrupt
label:	To transfer control to the command at "label"

When you use the onintr command without any option, the default action of process termination takes place when you press <Ctrl+C> while the process is running. Thus, using the onintr command without any option is redundant. Here, we enhance the script in the **while_demo** file in Chapter W29, so that you cannot terminate execution of this program with <Ctrl+C>. The enhanced version is in the **onintr_demo** file, as shown in the following session. Note that the onintr command is used to transfer control to the command at the interrupt_label: label when you press <Ctrl+C> while executing this program. The code at this label is a goto command that transfers control to the onintr interrupt command to reset the interrupt handling capability of the code, effectively ignoring <Ctrl+C>. A sample run illustrates this point.

```
% cat onintr_demo
#!/bin/tcsh

# Intercept <Ctrl+C> and transfer control to the command at
backagain:
    onintr interrupt

# Set the secret code
set secretcode = agent007

# Get user input
echo "Guess the code\!"
echo -n "Enter your guess: "
set yourguess = `head -1`

# As long as the user input is not the secret code (agent007 in this case),
# loop here: display a message and take user input gain. When the user
# input matches the secret code, terminate the loop and execute the
# following echo command.
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
```

```
# Code executed when you press <Ctrl+C>
interrupt:
    echo "Nice try -- you cannot terminate me by <Ctrl+C>\!"
    goto backagain
% ./onintr_demo
Guess the code!
Enter your guess: codecracker
Good guess but wrong. Try again!
Enter your guess: <Ctrl+C>
Nice try -- you cannot terminate me by <Ctrl+C>!
Guess the code!
Enter your guess: Lionking
Good guess but wrong. Try again!
Enter your guess: agent007
Wow! You are a genius!
%
```

The net effect of using the `onintr` command in the preceding script is to ignore a keyboard interrupt. You can achieve the same effect by using the command with the – option. Thus, the whole interrupt handling code in the **onintr_demo** program can be replaced by the `onintr` – command; no code is needed at any label, but then the code does not display any message for you when you press `<Ctrl+C>`.

To terminate programs that ignore terminal interrupts, you have to use the `kill` command. You can do so by suspending the process by pressing `<Ctrl+Z>`, using the `ps` command to get the process ID (PID) of the process, and terminating it with the `kill` command.

You can modify the script in the **onintr_demo** file so that it ignores the keyboard interrupt, clears the display screen, and turns off the echo. Whatever you enter at the keyboard, then, is not displayed. Next, it prompts you for the code word twice. If you do not enter the same code word both times, it reminds you of your bad short-term memory and quits. If you enter the same code word, it clears the display screen and prompts you to guess the code word again. If you do not enter the original code word, the screen is cleared and you are prompted to guess again. The program does not terminate until you have entered the original code word. When you do enter it, the display screen is cleared, a message is displayed at the top left of the screen, and echo is turned on. Because the terminal interrupt is ignored, you cannot terminate the program by pressing `<Ctrl+C>`. The `stty -echo` command turns off the echo. Thus, when you type the original code word (or any guesses), it is not displayed on the screen. The `clear` command clears the display screen and locates the cursor at the top-left corner. The `stty echo` command turns on the echo. The resulting script is in the **canleave** file shown in the following session.

```
% cat canleave
#!/bin/tcsh

# File Name:       ~/linux2e/TCShell/canleave
# Author:          Syed Mansoor Sarwar
# Written:         August 18, 2004
# Modified:        May 8, 2004, January 15, 2017
# Purpose:         To allow a user to leave his/her terminal for a short duration
#                  of time by locking the terminal after taking a code from the
#                  user. Terminal is unlocked only when the user re-enters the
#                  same code. Ignores command line arguments. Does not terminate
#                  with <Ctrl+C>.
# Brief Description:
#          Clear screen and turn off echo (i.e., do not display what the user
```

```
#           types at the keyboard). Take user code, save it, and ask the user
#           to re-enter his/her code just to make sure that the user remembers
#           the code that he/she has entered. It is done twice. If the user does
#           not enter the same code, the program terminates after displaying a
#           message for the user. The user is prompted to enter the original
#           code. If the user enters the wrong code, the program keeps on
#           prompting the user until he/she enters the orignal code. The
#           keyboard is then unlocked, echo is turned on, and program exits,
#           allowing the user to use his/her system again.

# Ignore terminal interrupt
onintr -

# Clear the screen, locate the cursor at the top-left corner,
# and turn off echo
clear
stty -echo

# Set the secret code
echo -n "Enter your code word: "
set secretcode = `head -1`
echo " "

# To make sure that the user remembers the code word
echo -n "Enter your code word again: "
set same = `head -1`
echo " "
if ( $secretcode != $same ) then
    echo "Work on your short-term memory before using this code\!"
    exit 1
endif

# Keyboard locked. Hit <Enter> to continue.
clear
echo -n "Keyboard locked. Hit <Enter> to continue."
set ignore = `head -1`
clear

# Get user guess to unlock the terminal
echo -n "Enter the code word: "
set yourguess = `head -1`
echo " "

# As long as the user input is not the original code word, loop here: display
# a message and take user input gain. When the user input matches the secret
# code word, terminate the loop and execute the following echo command.
while ( "$secretcode" != "$yourguess" )
    clear
    echo -n "Enter the code word: "
    set yourguess = `head -1`
end

# Set terminal to echo mode
clear
echo "Back again\!"
stty echo
exit 0
%
```

You can use this script to lock your terminal before you leave it to pick up a printout or get a can of soda; hence, the name **canleave** (can leave). Using it saves you the time otherwise required for the logout and login procedures.

W30.6 DEBUGGING SHELL PROGRAMS

You can debug your TC shell scripts by using the `-x` (echo) option of the `csh` command. This option displays each line of the script after variable substitution but before execution. You can combine the `-x` option with the `-v` (verbose) option to display each command line of the script, as it appears in the script file before its execution. You can also invoke the `csh` command from the command line to run the script, or you can make it part of the script, as in `#!/bin/csh -xv`.

In the following session, we show how a shell script can be debugged. The script in the **debug_demo** file prompts you to enter a digit. If you enter a value between 1 and 9, it displays a message informing you that what you entered was good and quits. If you enter any other value, it informs you accordingly and exits. When we execute the script and enter `4`, a valid input, it displays the message `var1cat: Undefined variable.` Similarly, when we run the script and enter `10`, an invalid input, we get the same error message.

```
% cat debug_demo
#!/bin/csh

echo -n "Enter a digit: "
set var1 = `head -1`
if (("$var1" >= 1) && ("$var1" <= 9)) then
   echo "Good input is $var1!!"
else
   echo "Bad input is $var1!!"
endif
exit 0
% ./debug_demo
Enter a digit: 4
var1cat: Undefined variable.
% ./debug_demo
Enter a digit: 10
var1cat: Undefined variable.
%
```

We debug the program by using the `csh -xv debug_demo` command. The last two lines of output of the runtime trace show the problem area. Somehow, the `$var1` variable is followed by the character sequence `cat canleave`. We put a space between `$var1` and `!!` at the end of the `echo` command. This helps a little in that the output becomes `Good input is 4 cat canleave`. We then realize that the problem is with the two back-to-back bang signs (`!!`) at the end of the two `echo` commands. The bang sign is has a special meaning in the TC shell that indicates the beginning of a previous event; `!!` means the event immediately preceding. This means that `cat canleave` was the command that was executed prior to this command. The problem is taken care of by either escaping one of the two `!` signs, as in `$var1\!!`, or by removing one of the `!` signs, as in `$var1!`. After we take care of this problem, the script works properly.

```
% csh -xv debug_demo


echo -n "Enter a digit: "
echo -n Enter a digit:
Enter a digit: set var1 = `head -1`
```

```
set var1 = `head -1`
head -1
4
if ( ( "$var1" > = 1 ) && ( "$var1" < = 9 ) ) then
if ( ( 4 > = 1 ) && ( 4 < = 9 ) ) then
echo "Good input is $var1cat canleave"
var1cat: Undefined variable.
%
```

For larger scripts, it may become difficult to identify the problem area. In such cases, you should use the `echo` commands at different places in your script to narrow down on the problematic code region. It is similar to using the `cout` or `printf` statements in C, C++, and Java programs while debugging your code in these high-level languages.

The following in-chapter exercise has been designed to enhance your understanding of the interrupt processing and debugging features of the TC shell.

Exercise W30.4

Test the scripts in the **onintr_demo** and **canleave** files on your Linux system. Do they work as expected? Be sure that you understand them. If your versions do not work properly, use the `csh -xv` command to debug them.

SUMMARY

The TC shell has the built-in capability for numeric integer data processing in terms of arithmetic, logic, and shift operations. Combined with the array processing feature of the language, this allows the programmer to write useful programs for processing large datasets with relative ease. The numeric variables can be declared and processed by using the `@` and `set` commands.

The here document feature of the TC shell allows standard input of a command in a script to be attached to data within the script. The use of this feature results in more efficient programs. The reason is that no extra file-related operations, such as file open and read, are needed, as the data is within the script file and has probably been loaded into the main memory when the script was loaded.

The TC shell also allows the user to write programs that ignore signals such as terminal interrupt (`<Ctrl+C>`). This useful feature can be used, among other things, to disable program termination when it is in the middle of updating a file. The `onintr` command can be used to invoke this feature.

The TC shell programs can be debugged by using the `-x` and `-v` options of the `csh` command, as in `csh -xv filename`. This technique allows viewing of the commands in the user's script after variable substitution but before execution.

QUESTIONS AND PROBLEMS

1. Is the `expr` command needed in the TC shell?
2. Modify the `num_array_demo` script in Section W30.3 so that it takes the numbers to be added as the command-line arguments. Use the `while` control structure and integer arrays.

3. What is the difference between the following two commands if `students` is an array? See the second shell session in Section W30.3 for the current value of the `students` array. Explain your answer.

   a. `set Linux_Authors = $students[1-2]`

   b. `set Linux_Authors = ( $students[1-2] )`

4. Draw the diagram for the 11-element `students` array discussed in Section W30.3.

5. Modify the `fs` script in Section W30.3 so that it displays the size of all symbolic links instead of ordinary files.

6. Modify the `fs` script in Section W30.3 so that instead of displaying the size of all ordinary files, it displays the number of different types of files in the directory that is passed to it as a command-line argument: ordinary, directory, character special, block special, link, socket, and pipe.

7. Modify the `fs` script for Problem 6 so that it also displays the number of files that have the SUID bit set.

8. What is here document? Why is it useful?

9. The `dext` script in Section W30.4 takes a single name as a command-line argument. Modify this script so that it takes a list of names as command-line arguments. Use the `foreach` control structure to implement your solution.

10. The `canleave` script in Section W30.5 is designed to ignore keyboard interrupt. How can this program be terminated? Be precise.

11. Write a TC shell script that takes integer numbers as command-line arguments and displays their sum on the screen.

12. Write a TC shell script that takes an integer number from the keyboard and displays the Fibonacci numbers equal to the number entered from the keyboard. Thus, if the user enters 7, your script displays the first seven Fibonacci numbers.

13. What are signals in Linux? What types of signals can be intercepted in TC shell scripts?

14. Enhance the code of Problem 7 so that it cannot be terminated by pressing `<Ctrl+C>`. When the user presses `<Ctrl+C>`, your script should give a message to the user and continue.

15. Modify the script in Problem 11 so that it reads integers to be added as a here document.

16. Enhance the `onintr_demo` script so that it takes the code word and the category of the code word (e.g., scientist, TV newscaster, politician, celebrity, movie, sportsperson) as input. It then informs the user of the category and gives ten chances to the user to guess the code word.

17. Write a TC shell script that implements the following menu options:
    a. Display the name of the central processing unit (CPU) used by your system.
    b. Display the name of the operating system used by your computer.
    c. Display the results of a. and b. on the screen separated by a vertical tab.
    d. Display the full path names for the commands that have been executed on your system.
    e. Display the maximum number of files a process may open.
    f. Display the maximum number of simultaneous processes a user may have on the system.

    Your program should not terminate when the user presses `<Ctrl+C>`.

    Hint: Review the manual pages for the following commands: `hash`, `ulimit`, `uname`.

TABLE W30.1 Assignment Operators for the `@` Command

Operator	Meaning
`=`	Assigns the value of the expression on the right-hand side of = to the variable preceding it
`+=`	Adds the value of the expression on the right-hand side of = to the current value of the variable preceding it and assigns the result to the variable
`-=`	Subtracts the value of the expression on the right-hand side of = from the current value of the variable preceding it and assigns the result to the variable
`*=`	Multiplies the value of the expression on the right-hand side of = by the current value of the variable preceding it and assigns the result (product) to the variable
`/=`	Divides the value of the variable preceding = by the value of the expression on the right-hand side of = and assigns the result (quotient) to the variable
`%=`	Divides the value of the variable preceding = by the value of the expression on the right-hand side of = and assigns the remainder to the variable

TABLE W30.2 Some Important Signals, Their Numbers, and Their Purpose

Signal Name	Signal #	Purpose
SIGHUP (hang up)	1	Informs the process when the user who ran the process logs out and terminates the process
SIGINT (keyboard interrupt)	2	Informs the process when the user presses `<Ctrl+C>` and terminates the process
SIGQUIT (quit signal)	3	Informs the process when the user presses `<Ctrl+|>` or `<Ctrl+\>` and terminates the process
SIGKILL (sure kill)	9	Terminates the process when the user sends this signal to it with the `kill -9` command
SIGSEGV (segmentation violation)	11	Terminates the process upon memory fault when a process tries to access memory space that does not belong to it
SIGTERM (software termination)	15	Terminates the process when the kill command is used without any signal number
SIGTSTP (suspend/stop signal)	18	Suspends the process, usually `<Ctrl+Z>`
SIGCHLD (child finished execution)	20	Informs the process of the termination of one of its children