chapter

2

Shell Programming QuickStart

2.1 Taking a Peek at Shell Scripts

If you read, write, or maintain programs, the following samples will give you a quick overview of the construction and style of a shell script and introduce you to some of the constructs and syntax found in these programs. *Note: If you are not familiar with programming, skip this chapter and go to Chapter 3.* When you have finished learning how to write scripts, you may want to return to this chapter for a quick reference to refresh your memory.

The C shell and TC shell emulate the C language syntax whereas the Bourne shell is based on an older programming language called Algol.

The Bash and Korn shells tend to be a combination of both the Bourne and C shells, although these shells originated from the Bourne shell.

To illustrate the differences in the shells, four sample programs are provided, one for each shell (the C and TC shells are presented together here). Above each program, a list of basic constructs are described for the shell being examined.

2.2 Sample Scripts: Comparing the Major Shells

At the end of each section pertaining to a specific shell, you will find a small program to illustrate how to write a complete script. At first glance, the programs for each shell look very similar. They are. And they all do the same thing. The main difference is the syntax. After you have worked with these shells for some time, you will quickly adapt to the differences and start formulating your own opinions about which shell is your favorite. A detailed comparison of differences among the C/TC, Bourne, Bash, and Korn shells is found in Appendix B.

Before Getting Started. You must have a good handle on UNIX/Linux commands. If you do not know the basic commands, you cannot do much with shell programming. The next three chapters will teach you how to use some of the major UNIX/Linux commands, and Appendix A in the back of the book, gives you a list of the most common commands (also called utilities).

The Purpose. The sample scripts provided at the end of each section send a mail message to a list of users, inviting each of them to a party. The place and time of the party are set in variables. The list of guests is selected from a file called `guests`. The existence of the `guest` file is checked and if it does not exist, the program will exit. A list of foods is stored in a word list (array). A loop is used to iterate through the list of guests. Each user will receive an e-mail invitation telling him or her the time and place of the party and asking him or her to bring an item from the food list. A conditional is used to check for a user named `root`, and if he is on the guest list, he will be excluded; that is, he will not be sent an e-mail invitation. The loop will continue until the guest list is empty. Each time through the loop, a food item is removed from the list, so that each guest will be asked to bring a different food. If, however, there are more users than foods, the list is reset. This is handled with a standard loop control statement.

2.3 The C and TC Shell Syntax and Constructs

The basic C and TC shell syntax and constructs are listed in Table 2.1.

Table 2.1 C and TC Shell Syntax and Constructs

The shbang line	The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a hash mark #, an exclamation point ! (called a bang), followed by the full pathname of the shell, and any shell options. Any other lines beginning with a # are used as comments. **EXAMPLE** `#!/bin/csh or #!/bin/tcsh`
Comments	Comments are descriptive material preceded by a # sign; they are not executable statements. They are in effect until the end of a line and can be started anywhere on the line. **EXAMPLE** `# This is a comment`

Table 2.1 C and TC Shell Syntax and Constructs (continued)

Wildcards	There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or "wildcards." These characters are neither numbers nor letters. For example, the *, ?, and [ ] are used for filename expansion. The ! is the history character, the < , > , >> , <&, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted with a backslash or quote marks.
	EXAMPLE
	`rm *; ls ??;  cat file[1-3]; !!`
	`echo "How are you?"`
	`echo Oh boy\!`

Displaying output	To print output to the screen, the `echo` command is used. Wildcards must be escaped with either a backslash or matching quotes. **EXAMPLE** `echo "Hello to you\!"`

Local variables	Local variables are in scope for the current shell. When a script ends or the shell exits, they are no longer available; i.e., they go out of scope. Local variables are set and assigned values.
	EXAMPLE
	`set variable_name = value`
	`set name = "Tom Jones"`

Global variables	Global variables are called environment variables. They are set for the currently running shell and are available to any process spawned from that shell. They go out of scope when the script ends or the shell where they are defined exits.
	EXAMPLE
	`setenv  VARIABLE_NAME  value`
	`setenv PRINTER Shakespeare`

Extracting values from variables	To extract the value from variables, a dollar sign is used.
	EXAMPLE
	`echo $variable_name`
	`echo $name`
	`echo $PRINTER`

Reading user input	The special variable $< reads a line of input from the user and assigns it to a variable.
	EXAMPLE
	`echo "What is your name?"`
	`set name = $<`

Table 2.1 C and TC Shell Syntax and Constructs (continued)

Arguments	Arguments can be passed to a script from the command line. Two methods can be used to receive their values from within the script: positional parameters and the `argv` array.
	
	EXAMPLE
	
	`% scriptname arg1 arg2 arg3 ...`
	
	Using positional parameters:
	`echo $1 $2 $3` — `arg1` *is assigned to* `$1`, `arg2` *to* `$2`, *etc.*
	`echo $*` — *all the arguments*
	
	Using the argv array:
	`echo $argv[1] $argv[2] $argv[3]`
	`echo $argv[*]` — *all the arguments*
	`echo $#argv` — *the number of arguments*

Arrays	An array is a list of words separated by whitespace. The list is enclosed in a set of parentheses.
	
	The built-in `shift` command shifts off the left-hand word in the list.
	
	Unlike C, the individual words are accessed by index values, which start at 1 rather than 0.
	
	EXAMPLE
	
	`set word_list = ( word1 word2 word3 )`
	
	`set names = ( Tom Dick Harry Fred )`
	`shift names` — *removes* `Tom` *from the list*
	
	`echo $word_list[1]` — *displays first element of the list*
	`echo $word_list[2]` — *displays second element of the list*
	`echo $word_list or $word_list[*]` — *displays all elements of the list*
	`echo $names[1]`
	`echo $names[2]`
	`echo $names[3]`
	`echo $names or echo $names[*]`

Command substitution	To assign the output of a UNIX/Linux command to a variable, or use the output of a command in a string, the command is enclosed in backquotes. **EXAMPLE** `set variable_name=`command`` `echo $variable_name`  `set now = `date`` — *The command in backquotes is executed and its output* `echo $now` — *is assigned to the variable* `now` `echo "Today is `date`"` — *The output of the* `date` *command is inserted in the string*

Table 2.1 C and TC Shell Syntax and Constructs (continued)

Arithmetic	Variables that will hold the results of an arithmetic computation must be preceded by an `@` symbol and a space. Only integer arithmetic is provided by this shell.
	
	EXAMPLE
	
	`@ n = 5 + 5`
	`echo $n`

Operators	The C and TC shells support operators for testing strings and numbers similar to those found in the C language.
	
	EXAMPLE
	
	Equality:
	`==`
	`!=`
	
	Relational:
	`>` *greater than*
	`>=` *greater than or equal to*
	`<` *less than*
	`<=` *less than or equal to*
	
	Logical:
	`&&` *and*
	`||` *or*
	`!` *not*

Conditional statements	The `if` construct is followed by an expression enclosed in parentheses. The operators are similar to C operators. The `then` keyword is placed after the closing parentheses. An `if` must end with an `endif`. An alternative to `if/else if` is the `switch` statement.
	
	EXAMPLE
	
	The `if` construct is:
	`if ( expression ) then`
	`  block of statements`
	`endif`
	
	The `if/else` construct is:
	`if ( expression ) then`
	`  block of statements`
	`else`
	`  block of statements`
	`endif`
	
	The `if/else/else if` construct is:
	`if ( expression ) then`
	`  block of statements`
	`else if ( expression ) then`
	`  block of statements`
	`else if ( expression ) then`
	`  block of statements`
	`else`
	`  block of statements`
	`endif`

Table 2.1 C and TC Shell Syntax and Constructs (continued)

Conditional statements *(continued)*	The switch construct is:
	`switch variable_name`
	`  case constant1:`
	`  statements`
	`  case constant2:`
	`  statements`
	`  case constant3:`
	`  statements`
	`  default:`
	`  statements`
	`endsw`
		`switch ( "$color" )`
		`  case blue:`
		`  echo $color is blue`
		`  breaksw`
		`  case green:`
		`  echo $color is green`
		`  breaksw`
		`  case red:`
		`  case orange:`
		`  echo $color is red or orange`
		`  breaksw`
		`  default:`
		`  echo "Not a valid color"`
		`endsw`

Loops	There are two types of loops; the `while` and `foreach` loop. The `while` loop is followed by an expression enclosed in parentheses, a block of statements, and terminated with the end keyword. As long as the expression is true, the looping continues. The `foreach` loop is followed by a variable name and a list of words enclosed in parentheses, a block of statements, and terminates with the end keyword. The foreach loop iterates through a list of words, processing a word and then shifting it off, then moving to the next word. When all words have been shifted from the list, it ends. The loop control commands are `break` and `continue`.	

	EXAMPLE
	`while ( expression )`
	`  block of statements`
	`end`
	
	`foreach variable ( word list )`
	`  block of statements`
	`end`
	`------------------------------`
	`foreach color (red green blue)`
	`  echo $color`
	`end`	

File testing	The C shell has a built-in set of options for testing attributes of files, such as whether it is a directory, a plain file (not a directory), a readable file, and so forth. For other types of file tests, the UNIX `test` command is used. See Example 2.1 for a demonstration.	

Table 2.1 C and TC Shell Syntax and Constructs (continued)

File testing *(continued)*	**EXAMPLE** `-r` *Current user can read the file* `-w` *Current user can write to the file* `-x` *Current user can execute the file* `-e` *File exists* `-o` *Current user owns the file* `-z` *File is zero length* `-d` *File is a directory* `-f` *File is a plain file*

EXAMPLE 2.1

```
    #!/bin/csh -f
1   if ( -e file ) then
        echo file exists
    endif

2   if ( -d file ) then
        echo file is a directory
    endif

3   if ( ! -z file ) then
        echo file is not of zero length
    endif

4   if ( -r file && -w file ) then
        echo file is readable and writable
    endif
```

2.3.1 The C/TC Shell Script

The program in Example 2.2 is an example of a C shell/TC shell script. The program contains many of the constructs discussed in Table 2.1.

EXAMPLE 2.2

```
1   #!/bin/csh -f
2   # The Party Program--Invitations to friends from the "guest" file
3   set guestfile = ~/shell/guests
4   if ( ! -e "$guestfile" ) then
        echo "$guestfile:t non-existent"
        exit 1
5   endif
6   setenv PLACE "Sarotini's"
```

EXAMPLE 2.2 (CONTINUED)

```
7   @ Time = `date +%H` + 1
8   set food = ( cheese crackers shrimp drinks "hot dogs" sandwiches )
9   foreach person ( `cat $guestfile` )
10      if ( $person =~ root ) continue
11      mail -v -s "Party" $person << FINIS   # Start of here document
        Hi $person! Please join me at $PLACE for a party!
        Meet me at $Time o'clock.
        I'll bring the ice cream. Would you please bring $food[1] and
        anything else you would like to eat? Let me know if you can
        make it. Hope to see you soon.
              Your pal,
              ellie@`hostname`       # or `uname -n`
12  FINIS
13      shift food
14      if ( $#food ==  0 ) then
              set food = ( cheese crackers shrimp drinks "hot dogs"
                           sandwiches )
        endif
15  end

    echo "Bye..."
```

EXPLANATION

1 This line lets the kernel know that you are running a C shell script. The `-f` option is a fast startup. It says, "Do not execute the `.cshrc` file," an initialization file that is automatically executed every time a new `csh` program is started.

2 This is a comment. It is ignored by the shell, but important for anyone trying to understand what the script is doing.

3 The variable `guestfile` is set to the full pathname of a file called `guests`.

4 This line reads: If the file `guests` does not exist, then print to the screen "`guests` non-existent" and exit from the script with an exit status of 1 to indicate that something went wrong in the program.

5 This marks the end of the statements based on the `if` condition.

6 Variables are assigned the values for the place and time. The `PLACE` variable is an environment variable.

7 The `Time` variable is a local variable. The `@` symbol tells the C shell to perform its built-in arithmetic; that is, add 1 to the `Time` variable after extracting the hour from the `date` command. The `Time` variable is spelled with an uppercase T to prevent the C shell from confusing it with one of its reserved words, `time`.

8 The `food` array is created. It consists of a list of words separated by whitespace. Each word is an element of the `food` array.

EXPLANATION (CONTINUED)

9 The `foreach` loop consists of a list, created by using command substitution, `` `cat $guestfile` ``. The output of the `cat` command will create a list of guests from a file. For each person on the guest list, except the user `root`, a mail message will be created inviting the person to a party at a given place and time, and asking him or her to bring one of the foods on the list.

10 The condition tests to see if the value of the variable, `person`, matches the word `root`. If it does, the `continue` statement causes control to go immediately back to the top of the loop (rather than executing any further statements). The next word on the list will be then be processed by the `foreach`.

11 The mail message is created in what is called a `here document`. All text from the user-defined word `FINIS` to the final `FINIS` will be sent to the `mail` program. The `foreach` loop shifts through the list of names, performing all of the instructions from the `foreach` to the keyword `end`.

12 `FINIS` is a user-defined terminator that ends the `here document`, which consists of the body of an e-mail message.

13 After a message has been sent, the food list is shifted to the left with the `shift` command, so that the next person on the guest list will get the next food item on the list.

14 If the food list is empty, it will be reset to ensure that any additional guests will be instructed to bring a food item.

15 This marks the end of the looping statements.

2.4 The Bourne Shell Syntax and Constructs

The basic Bourne shell syntax and constructs are listed in Table 2.2.

Table 2.2 Bourne Shell Syntax and Constructs

The shbang line	The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a `#!` followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell. **EXAMPLE** `#!/bin/sh`

Comments	Comments are descriptive material preceded by a `#` sign. They are in effect until the end of a line and can be started anywhere on the line.
	EXAMPLE
	`# this text is not`
	`# interpreted by the shell`

Table 2.2 Bourne Shell Syntax and Constructs (continued)

Wildcards	There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or "wildcards." These characters are neither numbers nor letters. For example, the *, ?, and [ ] are used for filename expansion. The <, >, 2>, >>, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.
	
	EXAMPLE
	
	Filename expansion:
	`rm *; ls ??;  cat file[1-3];`
	
	Quotes protect metacharacter:
	`echo "How are you?"`

Displaying output	To print output to the screen, the `echo` command is used. Wildcards must be escaped with either a backslash or matching quotes. **EXAMPLE** `echo "What is your name?"`

Local variables	Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables are set and assigned values.
	
	EXAMPLE
	
	`variable_name=value`
	`name="John Doe"`
	`x=5`

Global variables	Global variables are called environment variables. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.
	
	EXAMPLE
	
	`VARIABLE_NAME=value`
	`export VARIABLE_NAME`
	
	`PATH=/bin:/usr/bin:.`
	`export PATH`

Extracting values from variables	To extract the value from variables, a dollar sign is used.
	
	EXAMPLE
	
	`echo $variable_name`
	`echo $name`
	`echo $PATH`

Table 2.2 Bourne Shell Syntax and Constructs (continued)

Reading user input	The `read` command takes a line of input from the user and assigns it to a variable(s) on the right-hand side. The `read` command can accept muliple variable names. Each variable will be assigned a word.
	
	EXAMPLE
	
	`echo "What is your name?"`
	`read name`
	`read name1 name2 ...`

Arguments (positional parameters)	Arguments can be passed to a script from the command line. Positional parameters are used to receive their values from within the script. **EXAMPLE** At the command line: `$ scriptname arg1 arg2 arg3 ...` In a script: `echo $1 $2 $3` *Positional parameters* `echo $*` *All the positional paramters* `echo $#` *The number of positional parameters*

Arrays (positional parameters)	The Bourne shell does support an array, but a word list can be created by using positional parameters. A list of words follows the built-in `set` command, and the words are accessed by position. Up to nine positions are allowed.
	
	The built-in `shift` command shifts off the first word on the left-hand side of the list.
	
	The individual words are accessed by position values starting at 1.
	
	EXAMPLE
	
	`set word1 word2 word3`
	`echo $1 $2 $3` *Displays* `word1`, `word2`, *and* `word3`
	
	`set apples peaches plums`
	`shift` *Shifts off* `apples`
	`echo $1` *Displays first element of the list*
	`echo $2` *Displays second element of the list*
	`echo $*` *Displays all elements of the list*

Command substitution	To assign the output of a UNIX/Linux command to a variable, or use the output of a command in a string, backquotes are used. **EXAMPLE** `variable_name=`command`` `echo $variable_name`  `now=`date`` `echo $now` `echo "Today is `date`"`

Table 2.2 Bourne Shell Syntax and Constructs (continued)

Arithmetic	The Bourne shell does not support arithmetic. UNIX/Linux commands must be used to perform calculations. **EXAMPLE** `n=`expr 5 + 5` `echo $n`
Operators	The Bourne shell uses the built-in `test` command operators to test numbers and strings. **EXAMPLE** Equality: `=` *string* `!=` *string* `-eq` *number* `-ne` *number* Logical: `-a` *and* `-o` *or* `!` *not* Relational: `-gt` *greater than* `-ge` *greater than, equal to* `-lt` *less than* `-le` *less than, equal to*

Conditional statements	The `if` construct is followed by a command. If an expression is to be tested, it is enclosed in square brackets. The `then` keyword is placed after the closing parenthesis. An `if` must end with a `fi`.
	
	EXAMPLE
	
	The `if` construct is:
	`if command`
	`then`
	`  block of statements`
	`fi`
	
	`if [ expression ]`
	`then`
	`  block of statements`
	`fi`
	
	The `if/else` construct is:
	`if [ expression ]`
	`then`
	`  block of statements`
	`else`
	`  block of statements`
	`fi`

Table 2.2 Bourne Shell Syntax and Constructs (continued)

Conditional statements *(continued)*	The if/else/else if construct is:
	`if command`
	`then`
	`  block of statements`
	`elif command`
	`then`
	`  block of statements`
	`elif command`
	`then`
	`  block of statements`
	`else`
	`  block of statements`
	`fi`
	`-------------------------`
	`if [ expression ]`
	`then`
	`  block of statements`
	`elif [ expression ]`
	`then`
	`  block of statements`
	`elif [ expression ]`
	`then`
	`  block of statements`
	`else`
	`  block of statements`
	`fi`
		The case command construct is:
		`case variable_name in`
		`  pattern1)`
		`  statements`
		`  ;;`
		`  pattern2)`
		`  statements`
		`  ;;`
		`  pattern3)`
		`  ;;`
		`  *) default value`
		`  ;;`
		`esac`
		
		`case "$color" in`
		`  blue)`
		`  echo $color is blue`
		`  ;;`
		`  green)`
		`  echo $color is green`
		`  ;;`
		`  red|orange)`
		`  echo $color is red or orange`
		`  ;;`
		`  *) echo "Not a color" # default`
		`esac`

Loops	There are three types of loops: while, until and for.
	
	The while loop is followed by a command or an expression enclosed in square brackets, a do keyword, a block of statements, and terminated with the done keyword. As long as the expression is true, the body of statements between do and done will be executed.
	
	The until loop is just like the while loop, except the body of the loop will be executed as long as the expression is false.
	
	The for loop used to iterate through a list of words, processing a word and then shifting it off, to process the next word. When all words have been shifted from the list, it ends. The for loop is followed by a variable name, the in keyword, and a list of words then a block of statements, and terminates with the done keyword.
	The loop control commands are break and continue.
	
	EXAMPLE
	
	`while command`
	`do`
	`  block of statements`
	`done`
	
	`while [ expression ]`
	`do`
	`  block of statements`
	`done`	

Table 2.2 Bourne Shell Syntax and Constructs (continued)

Loops *(continued)*	`until command`
	`do`
	`  block of statements`
	`done`
	
	`until [ expression ]`
	`do`
	`  block of statements`
	`done`
		`for variable in word1 word2 word3 ...`
		`do`
		`  block of statements`
		`done`

File testing	The Bourne shell uses the `test` command to evaluate conditional expressions and has a built-in set of options for testing attributes of files, such as whether it is a directory, a plain file (not a directory), a readable file, and so forth. See Example 2.3. **EXAMPLE** `-d` *File is a directory* `-f` *File exists and is not a directory* `-r` *Current user can read the file* `-s` *File is of nonzero size* `-w` *Current user can write to the file* `-x` *Current user can execute the file*	

EXAMPLE 2.3

```
    #!/bin/sh
1   if [ -f file ]
    then
        echo file exists
    fi

2   if [ -d file ]
    then
         echo file is a directory
    fi

3   if [ -s file ]
    then
         echo file is not of zero length
    fi

4   if [ -r file -a -w file ]
    then
         echo file is readable and writable
    fi
```

Table 2.2 Bourne Shell Syntax and Constructs (continued)

Functions	Functions allow you to define a section of shell code and give it a name. The Bourne shell introduced the concept of functions. The C and TC shells do not have functions.
	
	EXAMPLE
	
	`function_name() {`
	`  block of code`
	`}`
	
	`----------`
	
	`lister() {`
	`  echo Your present working directory is` `` `pwd` ``
	`  echo Your files are:`
	`  ls`
	`}`

2.4.1 The Bourne Shell Script

EXAMPLE 2.4

```
1   #!/bin/sh
2   # The Party Program--Invitations to friends from the "guest" file
3   guestfile=/home/jody/ellie/shell/guests
4   if [ ! -f "$guestfile" ]
5   then
6        echo "`basename $guestfile` non-existent"
7        exit 1
8   fi
9   PLACE="Sarotini's"; export PLACE
10  Time=`date +%H`
    Time=`expr $Time + 1`
11  set cheese crackers shrimp drinks "hot dogs" sandwiches
12  for person in `cat $guestfile`
    do
13       if [ $person =~ root ]
         then
14            continue
15       else
16       #     mail -v -s "Party" $person <<- FINIS
```

EXAMPLE 2.4 (CONTINUED)

```
17              cat <<-FINIS
                Hi ${person}! Please join me at $PLACE for a party!
                Meet me at $Time o'clock.
                I'll bring the ice cream. Would you please bring $1 and
                anything else you would like to eat? Let me know if you
                can make it. Hope to see you soon.
                    Your pal,
                    ellie@`hostname`
                FINIS
18              shift
19              if [ $# -eq 0 ]
                then
20                 set cheese crackers shrimp drinks "hot dogs" sandwiches
                fi
            fi
21      done
        echo "Bye..."
```

EXPLANATION

1 This line lets the kernel know that you are running a Bourne shell script.

2 This is a comment. It is ignored by the shell, but important for anyone trying to understand what the script is doing.

3 The variable `guestfile` is set to the full pathname of a file called `guests`.

4 This line reads: If the file `guests` does not exist, then print to the screen "`guests` non-existent" and exit from the script.

5 The `then` is usually on a line by itself, or on the same line as the `if` statement if it is preceded by a semicolon.

6 The UNIX `basename` command removes all but the filename in a search path. Because the command is enclosed in backquotes, command substitution will be performed and the output displayed by the `echo` command.

7 If the file does not exist, the program will exit. An exit with a value of 1 indicates that there was a failure in the program.

8 The `fi` keyword marks the end of the block of `if` statements.

9 Variables are assigned the values for the place and time. `PLACE` is an environment variable, because after it is set, it is exported.

10 The value in the `Time` variable is the result of command substitution; i.e., the output of the `date +%H` command (the current hour) will be assigned to `Time`.

11 The list of foods to bring is assigned to special variables (positional parameters) with the `set` command.

12 The `for` loop is entered. It loops through until each person listed in the guest file has been processed.

EXPLANATION (CONTINUED)

13 If the variable `person` matches the name of the user `root`, loop control will go to the top of the `for` loop and process the next person on the list. The user `root` will not get an invitation.

14 The `continue` statement causes loop control to start at line 12, rather than continuing to line 16.

15 The block of statements under `else` are executed if line 13 is not true.

16 The mail message is sent when this line is uncommented. It is a good idea to comment this line until the program has been thoroughly debugged, otherwise the e-mail will be sent to the same people every time the script is tested.

17 The next statement, using the `cat` command with the `here document`, allows the script to be tested by sending output to the screen that would normally be sent through the mail when line 7 is uncommented.

18 After a message has been sent, the food list is shifted so that the next person will get the next food on the list. If there are more people than foods, the food list will be reset, ensuring that each person is assigned a food.

19 The value of `$#` is the number of postional parameters left. If that number is 0, the food list is empty.

20 The food list is reset.

21 The `done` keyword marks the end of the block of statements in the body of the `for` loop.

2.5 The Korn Shell Constructs

The Korn and Bash shells are very similar. The following constructs will work for both shells. To see all the subtle variations, see the individual chapters for these shells.

Table 2.3 Korn Shell Syntax and Constructs

The shbang line	The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a `#!` followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell. **EXAMPLE** `#!/bin/ksh`
Comments	Comments are descriptive material preceded by a `#` sign. They are in effect until the end of a line and can be started anywhere on the line. **EXAMPLE** `# This program will test some files`

Table 2.3 Korn Shell Syntax and Constructs (continued)

Wildcards	There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or "wildcards." These characters are neither numbers nor letters. For example, the *, ?, and [ ] are used for filename expansion. The <, >, 2>, >>, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.
	EXAMPLE
	`rm *; ls ??;  cat file[1-3];`
	`echo "How are you?"`

Displaying output	To print output to the screen, the `echo` command can be used. Wildcards must be escaped with either a backslash or matching quotes. Korn shell also provides a built-in `print` function to replace the `echo` command.
	EXAMPLE
	`echo "Who are you?"`
	`print "How are you?"`

Local variables	Local variables are in scope for the current shell. When a script ends or the shell exits, they are no longer available; i.e., they go out of scope. The `typeset` built-in command can also be used to declare variables. Local variables are set and assigned values.
	EXAMPLE
	`variable_name=value`
	`typeset variable_name=value`
	`name="John Doe"`
	`x=5`

Global variables	Global variables are called environment variables. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends or the shell where they were defined exits.
	EXAMPLE
	`export VARIABLE_NAME =value`
	`export PATH=/bin:/usr/bin:.`

Extracting values from variables	To extract the value from variables, a dollar sign is used.
	EXAMPLE
	`echo $variable_name`
	`echo $name`
	`echo $PATH`

Reading user input	The user will be asked to enter input. The `read` command is used to accept a line of input. Multiple arguments to `read` will cause a line to be broken into words, and each word will be assigned to the named variable. The Korn shell allows the prompt and `read` command to be combined.

Table 2.3 Korn Shell Syntax and Constructs (continued)

Reading user input *(continued)*	**EXAMPLE**
	`read name?"What is your name?"` *The prompt is in quotes. After it is displayed, the* `read` *command waits for user input*
	`print -n "What is your name?"`
	`read name`
	`read name1 name2 ...`

Arguments	Arguments can be passed to a script from the command line. Positional parameters are used to receive their values from within the script. **EXAMPLE** At the command line: `$ scriptname arg1 arg2 arg3 ...` In a script: `echo $1 $2 $3` *Positional parameters,* `$1` *is assigned* `arg1`*,* `$2` *is assigned* `arg2`*, ...* `echo $*` *All the positional paramters* `echo $#` *The number of positional parameters*

Arrays	The Bourne shell utilizes positional parameters to create a word list. In addition to positional parameters, the Korn shell also supports an array syntax whereby the elements are accessed with a subscript, starting at 0. Korn shell arrays are created with the `set -A` command.
	EXAMPLE
	`set apples pears peaches` *Positional parameters*
	`print $1 $2 $3` `$1` *is* `apples`*,* `$2` *is* `pears`*,* `$3` *is* `peaches`
	`set -A array_name word1 word2 word3 ...` *Array*
	`set -A fruit apples pears plums`
	`print ${fruit[0]}` *Prints* `apples`
	`${fruit[1]} = oranges` *Assign a new value*

Arithmetic	The Korn shell supports integer arithmetic.The `typeset -i` command will declare an integer type variable. Integer arithmetic can be performed on variables declared this way. Otherwise, the `(( ))` syntax (`let` command) is used for arithmetic operations.
	EXAMPLE
	`typeset -i variable_name` *Declare integer*
	`typeset -i num` `num` *is declared as an integer*
	`num=5+4`
	`print $num` *Prints* `9`
	`(( n=5 + 5 ))` *The* `let` *command*
	`print $n` *Prints* `10`

Table 2.3 Korn Shell Syntax and Constructs (continued)

Command substitution	Like the C/TC shells and the Bourne shell, the output of a UNIX/Linux command can be assigned to a variable, or used as the output of a command in a string, by enclosing the command in backquotes. The Korn shell also provides a new syntax. Instead of placing the command between backquotes, it is enclosed in a set of parentheses, preceded by a dollar sign. **EXAMPLE** `variable_name=`command` `variable_name=$(command)` `echo $variable_name`  `echo "Today is `date`"` `echo "Today is $(date)"`
Operators	The Korn shell uses the built-in `test` command operators to test numbers and strings, similar to C language operators. **EXAMPLE** Equality: `=` *string, equal to* `!=` *string, not equal to* `==` *number, equal to* `!=` *number, not equal to* Relational: `>` *greater than* `>=` *greater than, equal to* `<` *less than* `<=` *less than, equal to* Logical: `&&` *and* `||` *or* `!` *not*

Conditional statements	The `if` construct is followed by an expression enclosed in parentheses. The operators are similar to C operators. The `then` keyword is placed after the closing parenthesis. An `if` must end with a `fi`. The new test command `[[ ]]` is now used to allow pattern matching in conditional expressions. The old test command `[ ]` is still available for backward compatibility with the Bourne shell. The `case` command is an alternative to `if/else`.
	
	EXAMPLE
	
	The `if` construct is:
	`if command`
	`then`
	`  block of statements`
	`fi`
	`----------`
	`if [[ string expression ]]`
	`then`
	`  block of statements`
	`fi`
	
	`----------`
	`if (( numeric expression ))`
	`then`
	`  block of statements`
	`fi`

Table 2.3 Korn Shell Syntax and Constructs (continued)

Conditional statements *(continued)*	The if/else construct is:
	`if command`
	`then`
	`  block of statements`
	`else`
	`  block of statements`
	`fi`
	`-------------------------`
	`if [[ expression ]]`
	`then`
	`  block of statements`
	`else`
	`  block of statements`
	`fi`
	`---------------------------`
	`if (( numeric expression ))`
	`then`
	`  block of statements`
	`else`
	`  block of statements`
	`fi`
	
	The case construct is:
	`case variable_name in`
	`  pattern1)`
	`  statements`
	`  ;;`
	`  pattern2)`
	`  statements`
	`  ;;`
	`  pattern3)`
	`  ;;`
	`esac`
	`-------------------------`
	`case "$color" in`
	`  blue)`
	`  echo $color is blue`
	`  ;;`
	`  green)`
	`  echo $color is green`
	`  ;;`
	`  red|orange)`
	`  echo $color is red or orange`
	`  ;;`
	`esac`
		The if/else/else if construct is:
		`if command`
		`then`
		`  block of statements`
		`elif command`
		`then`
		`  block of statements`
		`elif command`
		`then`
		`  block of statements`
		`else`
		`  block of statements`
		`fi`
		`---------------------------`
		`if [[ string expression ]]`
		`then`
		`  block of statements`
		`elif [[ string expression ]]`
		`then`
		`  block of statements`
		`elif [[ string expression ]]`
		`then`
		`  block of statements`
		`else`
		`  block of statements`
		`fi`
		`---------------------------`
		`if (( numeric expression ))`
		`then`
		`  block of statements`
		`elif (( numeric expression ))`
		`then`
		`  block of statements`
		`elif (( numeric expression ))`
		`then`
		`  block of statements`
		`else`
		`  block of statements`
		`fi`

Table 2.3 Korn Shell Syntax and Constructs (continued)

Loops	There are four types of loops: `while`, `until`, `for`, and `select`.
	
	The `while` loop is followed by an expression enclosed in square brackets, a `do` keyword, a block of statements, and terminated with the `done` keyword. As long as the expression is true, the body of statements between `do` and `done` will be executed.
	
	The `until` loop is just like the `while` loop, except the body of the loop will be executed as long as the expression is false.
	
	The `for` loop is used to iterate through a list of words, processing a word and then shifting it off, to process the next word. When all words have been shifted from the list, it ends.
	
	The `select` loop is used to provide a prompt (PS3 variable) and a menu of numbered items from which the user inputs a selection The input will be stored in the special built-in `REPLY` variable. The `select` loop is normally used with the `case` command.
	
	The loop control commands are `break` and `continue`. The `break` command allows control to exit the loop before reaching the end of it; the `continue` command allows control to return to the looping expression before reaching the end.
	
	EXAMPLE
	
	`while command`
	`do`
	`  block of statements`
	`done`
	`----------`
	`while [[ string expression ]]`
	`do`
	`  block of statements`
	`done`
	`----------`
	`while (( numeric expression ))`
	`do`
	`  block of statements`
	`done`
	
	`until command`
	`do`
	`  block of statements`
	`done`
	`----------`
	`until [[ string expression ]]`
	`do`
	`  block of statements`
	`done`
	`----------`
	`until (( numeric expression ))`
	`do`
	`  block of statements`
	`done`
	
	`for variable in word_list`
	`do`
	`  block of statements`
	`done`
	`----------`
	`for name in Tom Dick Harry`
	`do`
	`  print "Hi $name"`
	`done`
	
	`select variable in word_list`
	`do`
	`  block of statements`
	`done`
	`----------`
	`PS3="Select an item from the menu"`
	`for item in blue red green`
	`  echo $item`
	`done`
	
	Shows menu:
	1) blue
	2) red
	3) green

Table 2.3 Korn Shell Syntax and Constructs (continued)

File testing

The Korn shell uses the `test` command to evaluate conditional expressions and has a built-in set of options for testing attributes of files, such as whether it is a directory, a plain file (not a directory), a readable file, and so forth. See Example 2.5.

EXAMPLE

`-d`	*File is a directory*
`-a`	*File exists and is not a directory*
`-r`	*Current user can read the file*
`-s`	*File is of nonzero size*
`-w`	*Current user can write to the file*
`-x`	*Current user can execute the file*

EXAMPLE 2.5

```
    #!/bin/sh
1   if [ -a file ]
    then
         echo file exists
    fi

2   if [ -d file ]
    then
          echo file is a directory
    fi

3   if [ -s file ]
    then
          echo file is not of zero length
    fi

4   if [ -r file -a -w file ]
    then
          echo file is readable and writable
    fi
```

Functions

Functions allow you to define a section of shell code and give it a name. There are two formats: one from the Bourne shell, and the Korn shell version that uses the `function` keyword.

EXAMPLE

```
function_name() {
   block of code
}

function function_name {
   block of code
}
-------------------------
```

Table 2.3 Korn Shell Syntax and Constructs (continued)

Functions *(continued)*	`function lister {`
	`  echo Your present working directory is` `` `pwd` ``
	`  echo Your files are:`
	`  ls`
	`}`

2.5.1 The Korn Shell Script

EXAMPLE 2.6

```
1   #!/bin/ksh
2   # The Party Program--Invitations to friends from the "guest" file
3   guestfile=~/shell/guests
4   if [[ ! -a "$guestfile" ]]
    then
        print "${guestfile##*/} non-existent"
        exit 1
    fi
5   export PLACE="Sarotini's"
6   (( Time=$(date +%H) + 1 ))
7   set -A foods cheese crackers shrimp drinks "hot dogs" sandwiches
8   typeset -i n=0
9   for person in $(< $guestfile)
    do
10      if  [[ $person = root ]]
        then
              continue
        else
              # Start of here document
11            mail -v -s "Party" $person <<- FINIS
              Hi ${person}! Please join me at $PLACE for a party!
              Meet me at $Time o'clock.
              I'll bring the ice cream. Would you please bring
              ${foods[$n]} and anything else you would like to eat? Let
              me know if you can make it.
                    Hope to see you soon.
                              Your pal,
                              ellie@`hostname`
              FINIS
12            n=n+1
```

EXAMPLE 2.6 (CONTINUED)

```
            if (( ${#foods[*]} == $n ))
            then
14             set -A foods cheese crackers shrimp drinks "hot dogs"
               sandwiches
            fi
         fi
15 done
   print "Bye..."
```

EXPLANATION

1 This line lets the kernel know that you are running a Korn shell script.

2 This is a comment. It is ignored by the shell, but important for anyone trying to understand what the script is doing.

3 The variable `guestfile` is set to the full pathname of a file called `guests`.

4 This line reads: If the file `guests` does not exist, then print to the screen "`guests nonexistent`" and exit from the script.

5 An environment variable is assigned a value and exported (made available to subshells).

6 The output of the UNIX/Linux command, the hour of the day, is assigned to the variable called `Time`. Variables are assigned the values for the place and time.

7 The list of foods to bring is assigned to an array called `foods` with the `set -A` command. Each item on the list can be accessed with an index starting at 0.

8 The `typeset -i` command is used to create an integer value.

9 For each person on the guest list a mail message will be created inviting the person to a party at a given place and time, and assigning a food from the list to bring.

10 The condition tests for the user `root`. If the user is `root`, control will go back to the top of the loop and assign the next user in the guest list to the variable, `person`.

11 The mail message is sent. The message body is contained in a `here document`.

12 The variable `n` is incremented by 1.

13 If the number of elements in the array is equal to the value of the variable, then the end of the array has been reached.

14 This marks the end of the looping statements.

2.6 The Bash Shell Constructs

The Korn and Bash shells are very similar, but there are some differences. The Bash constructs are listed in Table 2.4.

Table 2.4 Bash Shell Syntax and Constructs

The shbang line	The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a `#!` followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell. **EXAMPLE** `#!/bin/bash`
Comments	Comments are descriptive material preceded by a `#` sign. They are in effect until the end of a line and can be started anywhere on the line. **EXAMPLE** `# This is a comment`

Wildcards	There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or "wildcards." These characters are neither numbers or letters. For example, the `*`, `?`, and `[ ]` are used for filename expansion. The `<`, `>`, `2>`, `>>`, and `|` symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.
	EXAMPLE
	`rm *; ls ??;  cat file[1-3];`
	`echo "How are you?"`

Displaying output	To print output to the screen, the `echo` command is used. Wildcards must be escaped with either a backslash or matching quotes. **EXAMPLE** `echo "How are you?"`

Local variables	Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables can also be defined with the built-in `declare` function. Local variables are set and assigned values.
	EXAMPLE
	`variable_name=value`
	`declare variable_name=value`
	
	`name="John Doe"`
	`x=5`

Table 2.4 Bash Shell Syntax and Constructs (continued)

Global variables	Global variables are called environment variables and are created with the `export` built-in command. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.
	The built-in `declare` function with the `-x` option also sets an environment variable and marks it for export.
	EXAMPLE
	`export VARIABLE_NAME=value`
	`declare -x VARIABLE_NAME=value`
	`export PATH=/bin:/usr/bin:.`

Extracting values from variables	To extract the value from variables, a dollar sign is used.
	EXAMPLE
	`echo $variable_name`
	`echo $name`
	`echo $PATH`

Reading user input	The user will be asked to enter input. The `read` command is used to accept a line of input. Multiple arguments to `read` will cause a line to be broken into words, and each word will be assigned to the named variable.
	EXAMPLE
	`echo "What is your name?"`
	`read name`
	`read name1 name2 ...`

Arguments	Arguments can be passed to a script from the command line. Positional parameters are used to receive their values from within the script. **EXAMPLE** At the command line: `$ scriptname arg1 arg2 arg3 ...` In a script: `echo $1 $2 $3` *Positional parameters* `echo $*` *All the positional paramters* `echo $#` *The number of positional parameters*
Arrays	The Bourne shell utilizes positional parameters to create a word list. In addition to positional parameters, the Bash shell supports an array syntax whereby the elements are accessed with a subscript, starting at 0. Bash shell arrays are created with the `declare -a` command.

Table 2.4 Bash Shell Syntax and Constructs (continued)

Arrays *(continued)*	**EXAMPLE**
	
	`set apples pears peaches  (positional parameters)`
	`echo $1 $2 $3`
	
	`declare -a array_name=(word1 word2 word3 ...)`
	`declare -a fruit=( apples pears plums )`
	`echo ${fruit[0]}`

Command substitution	Like the C/TC shells and the Bourne shell, the output of a UNIX/Linux command can be assigned to a variable, or used as the output of a command in a string, by enclosing the command in backquotes. The Bash shell also provides a new syntax. Instead of placing the command between backquotes, it is enclosed in a set of parentheses, preceded by a dollar sign.
	
	EXAMPLE
	
	`variable_name=`command``
	`variable_name=$( command )`
	`echo $variable_name`
	
	`echo "Today is `date`"`
	`echo "Today is $(date)"`

Arithmetic	The Bash shells support integer arithmetic. The `declare -i` command will declare an integer type variable.The Korn shell's `typeset` command can also be use for backward compatibility. Integer arithmetic can be performed on variables declared this way. Otherwise the `(( ))` (`let` command) syntax is used for arithmetic operations.
	
	EXAMPLE
	
	`declare -i variable_name` *used for* `bash`
	`typeset -i variable_name` *can be used to be compatible with* `ksh`
	
	`(( n=5 + 5 ))`
	`echo $n`

Operators	The Bash shell uses the built-in `test` command operators to test numbers and strings, similar to C language operators. **EXAMPLE** Equality: `==` *equal to* `!=` *not equal to* Relational: `>` *greater than* `>=` *greater than, equal to* `<` *less than* `<=` *less than, equal to* Logical: `&&` *and* `||` *or* `!` *not*

Table 2.4 Bash Shell Syntax and Constructs (continued)

Conditional statements	The `if` construct is followed by an expression enclosed in parentheses. The operators are similar to C operators. The `then` keyword is placed after the closing paren. An `if` must end with an `endif`. The new `[[ ]]` test command is now used to allow pattern matching in conditional expressions. The old `[ ]` test command is still available for backward compatibility with the Bourne shell. The `case` command is an alternative to `if/else`.

EXAMPLE

The `if` construct is:

```
if  command
then
   block of statements
fi

if  [[ expression  ]]
then
   block of statements
fi

if  (( numeric expression  ))
then
   block of statements
else
   block of statements
 fi
```

The `if/else` construct is:

```
if  command
then
   block of statements
else
   block of statements
fi

if  [[ expression ]]
then
   block of statements
else
   block of statements
fi

if  ((  numeric expression ))
then
   block of statements
else
   block of statements
fi
```

The `if/else/else if` construct is:

```
if  command
then
   block of statements
elif  command
then
   block of statements
else if  command
then
   block of statements
else
   block of statements
fi
-------------------------
if  [[ expression ]]
then
   block of statements
elif  [[  expression ]]
then
   block of statements
else if  [[  expression ]]
then
   block of statements
else
   block of statements
fi
-------------------------
if  ((  numeric expression ))
then
   block of statements
elif  ((  numeric expression ))
then
   block of statements
else if  ((numeric expression))
then
   block of statements
else
   block of statements
fi
```

Table 2.4 Bash Shell Syntax and Constructs (continued)

Conditional statements *(continued)*	The `case` construct is:
	`case variable_name in`
	`  pattern1)`
	`  statements`
	`  ;;`
	`  pattern2)`
	`  statements`
	`  ;;`
	`  pattern3)`
	`  ;;`
	`esac`
	
	`case "$color" in`
	`  blue)`
	`  echo $color is blue`
	`  ;;`
	`  green)`
	`  echo $color is green`
	`  ;;`
	`  red|orange)`
	`  echo $color is red or orange`
	`  ;;`
	`  *) echo "Not a matach"`
	`  ;;`
	`esac`

Loops	There are four types of loops: `while`, `until`, `for`, and `select`. The `while` loop is followed by an expression enclosed in square brackets, a `do` keyword, a block of statements, and terminated with the `done` keyword. As long as the expression is true, the body of statements between `do` and `done` will be executed. The compound test operator `[[ ]]` is new with Bash, and the old-style test operator `[ ]` can still be used to evaluate conditional expressions for backward compatibility with the Bourne shell. The `until` loop is just like the `while` loop, except the body of the loop will be executed as long as the expression is false. The `for` loop is used to iterate through a list of words, processing a word and then shifting it off, to process the next word. When all words have been shifted from the list, it ends. The `for` loop is followed by a variable name, the `in` keyword, a list of words, then a block of statements, and terminates with the `done` keyword. The `select` loop is used to provide a prompt and a menu of numbered items from which the user inputs a selection. The input will be stored in the special built-in `REPLY` variable. The `select` loop is normally used with the `case` command. The loop control commands are `break` and `continue`. The `break` command allows control to exit the loop before reaching the end of it, and the `continue` command allows control to return to the looping expression before reaching the end.

Table 2.4 Bash Shell Syntax and Constructs (continued)

Loops *(continued)*

EXAMPLE

```
while command
do
   block of statements
done
-------------------------
while [[ string expression ]]
do
   block of statements
done
-------------------------
while (( numeric expression ))
do
   block of statements
done
```

```
until command
do
   block of statements
done
-------------------------
until [[ string expression ]]
do
   block of statements
done
-------------------------
until (( numeric expression ))
do
   block of statements
done
```

```
for variable in word_list
do
   block of statements
done
-------------------------
for color in red green blue
do
   echo $color
done
```

```
select variable in word_list
do
   block of statements
done
-------------------------
PS3="Select an item from the menu"
do item in blue red green   Shows menu:
   echo $item                  1) blue
done                           2) red
                               3) green
```

Functions

Functions allow you to define a section of shell code and give it a name. There are two formats, one from the Bourne shell, and the Bash version that uses the `function` keyword.

EXAMPLE

```
function_name() {
   block of code
}

function  function_name {
   block of code
}
-----------------------
function  lister {
   echo Your present working directory is `pwd`
   echo Your files are:
   ls
}
```

2.6.1 The Bash Shell Script

EXAMPLE 2.7

```
1   #!/bin/bash
    # GNU bash versions 2.x
2   # The Party Program--Invitations to friends from the "guest" file
3   guestfile=~/shell/guests
4   if [[ ! -e "$guestfile" ]]
    then
5       printf "${guestfile##*/} non-existent"
        exit 1
    fi
6   export PLACE="Sarotini's"
7   (( Time=$(date +%H) + 1 ))
8   declare -a foods=(cheese crackers shrimp drinks `"hot dogs"` sandwiches)
9   declare -i  n=0
10  for person in $(cat $guestfile)
    do
11      if  [[ $person == root ]]
        then
              continue
        else
              # Start of here document
12            mail -v -s "Party" $person <<- FINIS
              Hi $person! Please join me at $PLACE for a party!
              Meet me at $Time o'clock.
              I'll bring the ice cream. Would you please bring
              ${foods[$n] and anything else you would like to eat?
              Let me know if you can make it.
                    Hope to see you soon.
                         Your pal,
                         ellie@$(hostname)
              FINIS
13            n=n+1
14            if (( ${#foods[*]} ==  $n ))
              then
15               declare -a foods=(cheese crackers shrimp drinks `"hot dogs"`
                                  sandwiches)
16            n=0
              fi
        fi
17  done
    printf "Bye..."
```

EXPLANATION

1 This line lets the kernel know that you are running a Bash shell script.

2 This is a comment. It is ignored by the shell, but important for anyone trying to understand what the script is doing.

3 The variable `guestfile` is set to the full pathname of a file called `guests`.

4 This line reads: If the file `guests` does not exist, then print to the screen "`guests non-existent`" and exit from the script.

5 The built-in `printf` function dipslays only the filename (pattern matching) and the string "`non-existent`".

6 An environment (global) variable is assigned and exported.

7 A numeric expression uses the output of the UNIX/Linux `date` command to get the current hour. The hour is assigned to the variable, `Time`.

8 A Bash array, `foods`, is defined (`declare -a`) with a list of elements.

9 An integer, `n`, is defined with an initial value of zero.

10 For each person on the guest list, except the user `root`, a mail message will be created inviting the person to a party at a given place and time, and assigning a food from the list to bring.

11 If the value in `$person` is `root`, control goes back to the top of the `for` loop and starts at the next person on the list.

12 The mail message is sent. The message body is contained in a `here document`.

13 The integer, `n`, is incremented by 1.

14 If the number of foods is equal to the value of the last number in the array index, the list is empty.

15 The array called `foods` is reassigned values. After a message has been sent, the food list is shifted so that the next person will get the next food on the list. If there are more people than foods, the food list will be reset, ensuring that each person is assigned a food.

16 The variable `n`, which will serve as the array index, is reset back to zero.

17 This marks the end of the looping statements.