CHAPTER 4

Execution Control

Chapter Objectives

This chapter discusses techniques for changing the flow of control in a program, which may be necessary for two reasons:

- You may want to execute some parts of the code under certain circumstances only
- You may want to repeat a section of code a certain number of times

In Chapter 3 we used the array notation to gather numbers into a form where they could be processed collectively rather than individually. This chapter deals with ***code blocks*** (collections of one or more lines of code) that solve a particular segment of a problem in the same way. We will see how to define a code block, how to decide to execute a code block under certain conditions only, and how to repeat execution of a code block.

4.1 Concept: Code Blocks

Some languages identify code blocks by enclosing them in braces (`{. . .}`); others identify them by the level of indentation of the text. The MATLAB language uses the occurrence of key command words in the text to define the extent of code blocks. Keywords like `if`, `switch`, `while`, `for`, `case`, `otherwise`, `else`, `elseif`, and `end` are identified with blue coloring by the MATLAB text editor. They are not part of the code block, but they serve as instructions on what to do with the code block and as delimiters that define the extent of the code block.

4.2 Conditional Execution in General

To this point, the statements written in our scripts (single code blocks) have been executed in sequence from the instruction at the top to the instruction at the bottom. However, it is frequently necessary to make choices about how to process a set of data based on some characteristic of that data. We have seen logical expressions that result in a Boolean result—`true` or `false`. This section discusses the code that implements the idea shown in Figure 4.1.

In the flowchart shown in Figure 4.1, a set of statements (the code block to be executed) is shown as a rectangle, a decision point is shown as a diamond, and the flow of program control is indicated by arrows. When decision points are drawn, there will be at least two arrows leaving that symbol, each labeled with the reason one would take that path. This concept makes the execution of a code block conditional upon some test. If the result of the test is `true`, the code block is executed. Otherwise, the code block is omitted, and the instruction(s) after the end of that code block is executed next.

An important generalization of this concept is shown in Figure 4.2. Here the solution is generalized to permit the first code block to be implemented under the first condition as before. Now, however, if that

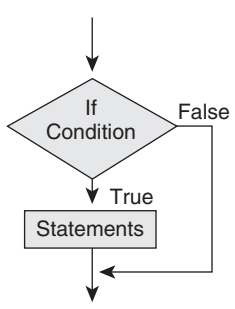

Figure 4.1 *Simple if statement*

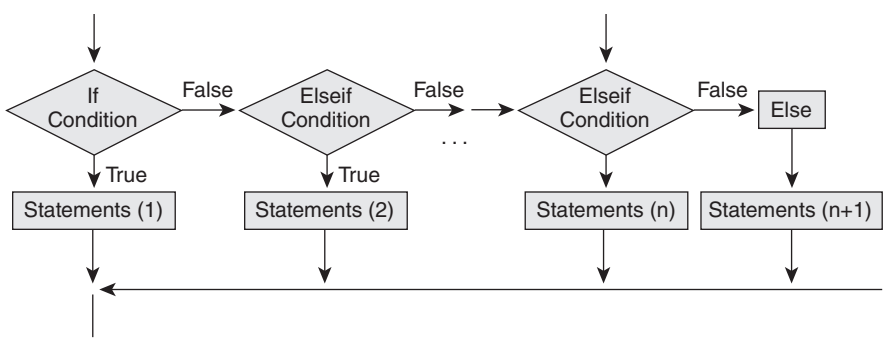

Figure 4.2 *Compound if statement*

first logical test returns `false`, a second test is performed to determine whether the second code block should be executed. If that test returns `false`, as many further tests as necessary may be performed, each with the appropriate code block to be implemented when the result is `true`. Finally, if none of these tests return true, the last code block, usually identified by the `else` keyword—(n + 1) in the figure—is executed. As the flowchart shows, as soon as one of the code blocks is executed, the next instruction to execute is the one that follows the conditional code after the `end` statement. In particular, if there is no `else` clause, it is possible that no code at all is executed in this conditional statement.

There are two common styles in which to implement this conditional behavior. First we will discuss the most general form, the `if` statement, and then we will discuss the more restrictive, but tidier, `switch` statement. Both implementations are found in most modern languages, albeit with slightly different syntax. In each case, the code block to be implemented is all the statements between the key words colored blue by the MATLAB editor.

4.3 `if` Statements

Here we introduce the concept of a programming template. Many programming texts still use the idea of **flowcharts**, such as those illustrated in Figures 4.1 and 4.2, to describe the design of a solution in a manner independent of the code implementation. However, since this graphical form cannot be maintained with a text editor, if the design of the solution changes, it is difficult to maintain any design description that is separate from the code itself.

Throughout the remainder of this text, we will describe the overall design of a code module using a **design template**. Design templates are a textual form of flowchart consisting of the key words that control program flow and placeholders that identify the code blocks and expressions that are

necessary to implement the solution logic. Design templates are powerful tools for the novice programmer to overcome the "blank sheet of paper" problem—"how do I start solving this problem?" All programmers need to do is recognize the nature of the solution and write down the appropriate template. Then solving a particular problem becomes the relatively simple task of writing the code blocks identified by the template.

To discuss the `if` statement, first we consider its general, language independent template and then its MATLAB implementation.

4.3.1 General Template

Template 4.1 shows the general template for the `if` statement. Note the following:

- The only essential ingredients are the first `if` statement, one code block, and the `end` statement. All other features may be added as the logic requires.
- The code blocks may contain any sequence of legal MATLAB statements, including other `if` statements (nested `if`s), `switch` statements, or iterations (see Section 4.5).
- Nested `if` statements with a code block are an alternative implementation of a logical `AND` statement.
- Recall that logical operations can be applied to a vector, resulting in a vector of Boolean values. This vector may be used as a logical expression. The `if` statement will accept this expression as `true` if all of the elements are `true`.

4.3.2 MATLAB Implementation

Listing 4.1 shows the MATLAB solution to a typical logical problem: determining whether a day is a weekday or a weekend day. It is assumed that the variable `day` is a number containing integer values from 1 to 7.

Template 4.1 General template for the `if` statement

```
if <logical expression 1>
     <code block 1>
elseif <logical expression 2>
     <code block 2>
     .
     .
     .
elseif <logical expression n>
     <code block n>
else
     <default code block>
End
```

Listing 4.1 `if` statement example

```
1. if day == 7          % Saturday
2. state = 'weekend'
3. elseif day == 1      % Sunday
4. state = 'weekend'
5. else
6. state = 'weekday'
7. end
```

In Listing 4.1:

Line 1: The first logical expression determines whether `day` is 7.

Line 2: The corresponding code block sets the value of the variable `state` to the string `'weekend'`. In general, there can be as many statements within a code block as necessary.

Line 3: The second logical expression determines whether `day` is 1.

Line 4: The corresponding code block also sets the value of the variable `state` to the string `'weekend'`.

Line 5: The key word `else` introduces the default code block executed when none of the previous tests pass.

Line 6: The default code block sets the value of the variable `state` to the string `'weekday'`.

Exercise 4.1 gives you the opportunity to practice using `if` statements, and Listing 4.2 shows a script that will satisfy Exercise 4.1.

Exercise 4.1 Using `if` statements

Write a script that uses `input(...)` to request a numerical grade in percentage and uses `if` statements to convert that grade to a letter grade according to the following table:

90% and better: A
80%–90%: B
70%–80%: C
60%–70%: D
Below 60%: F

Test your script by running it repeatedly for legal and illegal values of the grade percentage.

Check your work against the script shown in Listing 4.2.

Listing 4.2 Script with `if` statements

```
1. grade = input('what grade? ');
2. if grade >= 90
3.     letter = 'A'
4. elseif grade >= 80
5.     letter = 'B'
6. elseif grade >= 70
7.     letter = 'C'
8. elseif grade >= 60
9.     letter = 'D'
10. else
11.     letter = 'F'
12. end
```

In Listing 4.2:

Line 1: Requests a grade value from the user with the `input(...)` function. The prompt appears in the Command window, and the system waits for a line of text from the user and converts that line as it would any other Command window line, returning the result to the variable `grade`.

Line 2: The first logical expression looks for the grade that earns an A.

Line 3: The corresponding code block sets the value of the variable `letter` to `'A'`.

Lines 4–9: The corresponding logic for letter grades B, C, and D.

Lines 10–12: The default logic setting the variable `letter` to `'F'`.

4.3.3 Important Ideas

There are two important ideas that are necessary for the successful implementation of `if` statements: the general form of the logical expressions and short-circuit analysis.

Common Pitfalls 4.1

The MATLAB Command window echoes logical results as 1 (true) or 0 (false). In spite of this appearance, logical values are ***not*** numeric and should never be treated as if they were.

Logical Expressions The `if` statement requires a logical expression for its condition. A logical expression is any collection of constants, variables, and operators whose result is a Boolean `true` or `false` value.

Logical expressions can be created in the following ways:

- The value of a Boolean constant (e.g., `true` or `false`)
- The value of a variable containing a Boolean result (e.g., `found`)
- The result of a logical operation on two scalar quantities (e.g., `A > 5`)

- The result of logically negating a Boolean quantity using the unary negation operator (e.g., `~found`)
- The result of combining multiple scalar logical expressions with the operators `&&` or `||` (e.g., `A && B` or `A || B`)
- The results of the functions that are the logical equivalent of the `&&`, `||`, and ~ operators: `and(A, B)` `or(A, B)` and `not(A)`
- The results of other functions that operate on Boolean vectors: `any(...)` and `all(...)`

The result from `any(...)` will be `true` if any logical value in the vector is `true`. The result from `all(...)` will be `true` only if all logical values in the vector are `true`. The function `all(...)` is implicitly called if you supply a vector of logical values to the `if` statement, as shown in Listing 4.3.

In Listing 4.3:

Line 1: Makes the variable `A` a logical vector.

Line 2: Using this as a logical expression, internally converts this expression to `all(A)`.

Line 3: All the values of `A` are not true; therefore, the above code body does not execute.

Line 4: Now, all the elements of `A` are true.

Lines 5–6: If we repeat the test, the code body will now execute.

Short-Circuit Evaluation When evaluating a sequence of logical `&&` or `||`, MATLAB will stop processing when it finds the first result that makes all subsequent processing irrelevant. This concept is best illustrated by an example. Assume that `A` and `B` are logical results and you want to evaluate `A && B`. Since the result of this is `true` only if both `A` and `B` are `true`, if you evaluate `A` and the result is `false`, no value of `B` can change the outcome `A && B`. Therefore, there is no reason to evaluate any more components of a logical `and` expression once a `false` result has been found. Similarly, if you want `A || B`, if `A` is found to be true, you do not need to evaluate `B`. For example, suppose you want to test the *n*th element of a vector `v` using a variable `n`, and you are concerned that `n` might not be a legal index value.

Listing 4.3 The `if` statement with a logical vector

```
1. A = [true true false]
2. if A
     % will not execute
3. end
4. A(3) = true;
5. if A
     % will execute
6. end
```

The following code could be used:

```
if (n <= length(v)) && (v(n) > 0)
     % success!
end
```

If `n` were not a legal index, the indexed accessor `v(n)` would cause an error for attempting to reach beyond the end of the vector. However, by putting the test of `n` first, the short-circuit logic would not process the second part of the expression if the test of `n` failed.

4.4 `switch` Statements

A `switch` statement implements the logic shown in Figure 4.2 in a different programming style by allowing the programmer to consider a number of different cases for the value of one variable. First we consider the general, language-independent template for `switch` statements, and then its MATLAB implementation.

4.4.1 General Template

Template 4.2 shows the general template for the `switch` statement.

Note the following:

- All tests refer to the value of the same parameter
- `case` specifications may be either a single value or a set of parameters enclosed in braces `{ ... }`
- `otherwise` specifies the code block to be executed when none of the case values apply
- The code blocks may contain any sequence of legal MATLAB statements, including other `if` statements (nested `if`s), `switch` statements, or iterations

Template 4.2 General template for the `switch` statement

```
switch <parameter>
   case <case specification 1>
      <code block 1>
   case <case specification 2>
      <code block 2>
.
.
   case <case specification n>
      <code block n>
   otherwise
      <default code block>
end
```

4.4.2 MATLAB Implementation

Listing 4.4 shows the MATLAB implementation of a typical logical problem: determining the number of days in a month. It assumes that the value of `month` is 1 . . . 12, and `leapYear` is a `logical` variable identifying the current year as a leap year.

In Listing 4.4:

Line 1: All tests refer to the value of the variable `month`.

Line 2: This `case` specification is a cell array (See Chapter 7 for specifics) containing the indices of the months with 30 days.

Line 3: The code block extends from the `case` statement to the next control statement (`case`, `otherwise`, or `end`).

Line 5: This code block contains an `if` statement to deal with the February case. It presumes that a Boolean variable `leapYear` has been created to indicate whether this month is in a leap year.

Lines 10–11: Deal with the remaining months.

Line 13: A built-in MATLAB function that announces the error and terminates the script.

Style Points 4.1

The usual description of the logic suggests that the last case in Listing 4.4 could be the otherwise clause. However, that would prevent you from being able to detect bad month number values, as this code does.

Hint 4.1

The second parameter to the `input(...)` statement prevents MATLAB from attempting to parse the data provided, returning a string instead. Without that activity suppressed, if you enter the string `'yes'`, MATLAB will rush off looking for a variable by that name.

Style Points 4.2

The use of indentation is not required in the MATLAB language, and it has no significance with regard to syntax. However, the appropriate use of indentation greatly improves the legibility of code and you should use it. You have probably already noted that in addition to colorizing control statements, the text editor automatically places the control statements in the indented positions illustrated in Listings 4.3 and 4.4.

Listing 4.4 Example of a `switch` statement

```
1. switch month
2.    case {9, 4, 6, 11}
          % Sept, Apr, June, Nov
3.          days = 30;
4.    case 2              % Feb
5.        if leapYear
6.                days = 29;
7.            else
8.                days = 28;
9.            end
10.   case {1, 3, 5, 7, 8, 10, 12}
          % other months
11.               days = 31;
12.   otherwise
13.                 error('bad month index')
14. end
```

Exercise 4.2 Using the `switch` statement

Write and test the script in Listing 4.4 using `input(...)` to request a numerical month value.

You will need to preset a value for `leapYear`.

Test your script by running it repeatedly for legal and illegal values of the month.

Modify your script to ask whether the current year is a leap year. (It's best to ask only for February.) You could use code like the following:

```
ans = input('leap year (yes/no)', 's');
leapYear = (ans(1) == 'y');
```

Test this new script thoroughly.

Try this script without the second parameter to `input(...)`. Can you explain what is happening?

Modify the script again to accept the year rather than yes/no, and implement the logic to determine whether that year is a leap year.

Try using the `switch` statement in Exercise 4.2.

4.5 Iteration in General

Iteration allows controlled repetition of a code block. Control statements at the beginning of the code block specify the manner and extent of the repetition:

- The `for` loop is designed to repeat its code block a fixed number of times and largely automates the process of managing the iteration.
- The `while` loop is more flexible in character. In contrast to the fixed repetition of the `for` loop, its code block can be repeated a variable number of times, depending on the values of data being processed. It is much more of a "do-it-yourself" iteration kit.

The `if` and `switch` statements allow us to decide to skip code blocks based on conditions in the data. The `for` and `while` constructs allow us to repeat code blocks. Note, however, that the MATLAB language is designed to avoid iteration. Under most circumstances of processing numbers, the array processing operations built into the language make do-it-yourself loop constructs unnecessary.

4.6 `for` Loops

Figure 4.3 shows a simple `for` loop. The hexagonal shape illustrates the control of repetition. The repeated execution of the code block is performed under the control of a loop-control variable. It is first set to an initial value

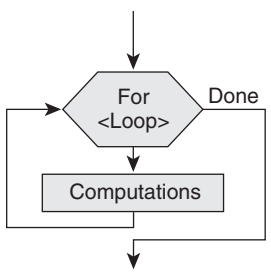

Figure 4.3 *Structure of a* `for` *loop*

that is tested against a terminating condition. If the terminating test succeeds, the program leaves the `for` loop. Otherwise, the computations in the code block are performed using the current value of that variable. When one pass through the code block is finished, the variable is updated to its next value, and control returns to the termination test.

4.6.1 General `for` Loop Template

The general template for implementing `for` loops is shown in Template 4.3. All of the mechanics of iteration control are handled automatically in the variable specification section. In some languages—especially those with their origins in C—the variable specification is a formidable collection of statements that provide great generality of loop management. The designers of the MATLAB language, with its origins in matrix processing, chose a much simpler approach for specifying the variable range, as shown in the general template. The repetition of the code block is managed completely by the specification of the loop control variable.

4.6.2 MATLAB Implementation

The core concept in the MATLAB `for` loop implementation is in the style of the variable specification, which is accomplished as follows:

```
<variable specification>: <variable> = <vector>
```

where `<variable>` is the name of the loop control variable and `<vector>` is any vector that can be created by the techniques discussed in Chapter 3. If

Template 4.3 General template for the `for` statement

```
for <variable specification>
      <code block>
end
```

we were to use the variable specification `x = A`, MATLAB would proceed as follows:

1. Set an invisible `index` to `1`.
2. Repeat steps 3 to 5 as long as that `index` is less than or equal to the length of `A`.
3. Set the value of `x` to `A(index)`
4. Evaluate the code block with that value of `x`
5. Increment the `index`

For a simple example of `for` loops, the code shown in Listing 4.5 solves a problem that should be done in a single MATLAB instruction: `max(A)` where `A` is a vector of integers. However, by expanding this into a `for` loop, we see the basic structure of the `for` loop at work.

In Listing 4.5:

Line 1: Creates a vector `A` with six elements.

Line 2: The tidiest way to find limits of a collection of numbers is to seed the result, `theMax`, with the first number.

Line 3: Iterates across the values of `A`.

Lines 4–6: The code block extends from the `for` statement to the associated `end` statement. The code will be executed the same number of times as the length of `A` *even if you change the value of* `x` *within the code block*. At each iteration, the value of `x` will be set to the next element from the array `A`.

Common Pitfalls 4.2

By setting the default answer to the first value, we avoid the problem of seeding the result with a value that could be already outside the range of the vector values. For example, we might think that `theMax = 0;` would be a satisfactory seed. However, this would not do well if all the elements of `A` were negative.

Line 8: The `fprintf(...)function` is a very flexible means of formatting output to the Command window. See the discussion in Chapter 8, or enter the following in the Command window:

```
> help fprintf
```

Listing 4.5 Example of a `for` statement

```
1. A = [6 12 6 91 13 6] % initial vector
2. theMax = A(1);       % set initial max value
3. for x = A            % iterate through A
4.     if x > theMax    % test each element
5.         theMax = x;
6.     end
7. end
8. fprintf('max(A) is %d\n', theMax);
```

4.6.3 Indexing Implementation

The above `for` loop implementation may seem very strange to those with a C-based language background, in which the loop-control variable is usually an index into the array being traversed rather than an element from that array. In order to illustrate the difference, we will adapt the code from Listing 4.5 to solve a slightly different problem that approximates the behavior of `max(A)`. This time we need to know not only the maximum value in the array, but also its index. This requires that we resort to indexing the array in a more conventional style, as shown in Listing 4.6.

In Listing 4.6:

Line 1: Generalizes the creation of the vector A using the `rand(...)` function to create a vector with 10 elements each between 0 and 100. The `ceil(...)` function rounds each value up to the next higher integer.

Lines 2 and 3: Initialize `theMax` and `theIndex`.

Line 4: Creates an anonymous vector of indices from 1 to the length of `A` and uses it to define the loop-control variable, `index`.

Line 5: Extracts the appropriate element from `A` to operate with as before.

Lines 6 and 7: The same comparison logic as shown in Listing 4.5.

Line 8: In addition to saving the new `max` value, we save the index where it occurs.

Line 11: This is our first occurrence where a logical line of code extends beyond the physical limitations of a single line. Since MATLAB normally uses the end of the line to indicate the end of an operation, we use ellipses `(...)` to specify that the logic is continued onto the next line.

You can enter and run these scripts by following Exercise 4.3.

Listing 4.6 `for` statement using indexing

```
1. A = floor(rand(1,10)*100)
2. theMax = A(1);
3. theIndex = 1;
4. for index = 1:length(A)
5.     x = A(index);
6.     if x > theMax
7.         theMax = x;
8.         theIndex = index;
9.     end
10. end
11. fprintf('the max value in A is %d at %d\n', ...
12.             theMax, theIndex);
```

Exercise 4.3 Producing `for` statement results

Enter and run the scripts in Listings 4.5 and 4.6. They should each produce the following results:

```
A =
  6 12  6 91 13 61 26 22 71 54
the max value in A is 91 at 4
>>
```

4.6.4 Breaking out of a `for` Loop

If you are in a `for` loop and find a circumstance where you really do not want to continue iterating, the `break` statement will skip immediately out of the innermost containing loop. If you want to continue iterating but omit all further steps of the current iteration, you can use the `continue` statement.

4.7 `while` Loops

We use `while` loops in general to obtain more control over the number of times the iteration is repeated. Figure 4.4 illustrates the control flow for a `while` loop. Since the termination test is performed before the loop is entered, the loop control expression must be initialized to a state that will normally permit loop entry. It is possible that the code block is not executed at all—for example—if there is no data to process.

4.7.1 General `while` Template

Template 4.4 shows the general template for implementing `while` loops. The logical expression controlling the iteration is testing some state of the workspace; therefore, two things that were automatic in the `for` loop must be manually accomplished with the `while` loop: initializing the test and updating the workspace in the code block so that the test will eventually fail and the iteration will stop.

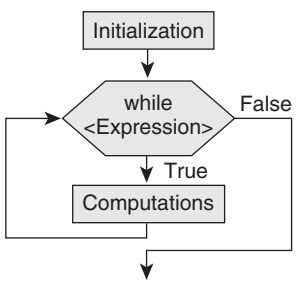

Figure 4.4 *Structure of a* `while` *loop*

Template 4.4 General template for the `while` statement

```
<initialization>
while <logical expression>
       <code block>   % must make some changes
              % to enable the loop to terminate
end
```

4.7.2 MATLAB `while` Loop Implementation

For the sake of consistency, Listing 4.7 shows you how to solve the same problem using the `while` syntax.

In Listing 4.7:

Lines 1–3: Create a test vector and initialize the answers as before.

Line 4: Initializes the index value since this is manually updated.

Line 5: This test will fail immediately if the vector `A` is empty.

Line 6: Extracts the item x from the array (good practice in general to clarify your code).

Lines 7–9: The same test as before to update the maximum value.

Line 11: "Manually" updates the index to move the loop closer to finishing.

Enter and run the script as described in Exercise 4.4.

Listing 4.7 `while` statement example

```
 1. A = floor(rand(1,10)*100)
 2. theMax = A(1);
 3. theIndex = 1;
 4. index = 1;
 5. while index <= length(A)
 6.     x = A(index);
 7.     if x > theMax
 8.         theMax = x;
 9.         theIndex = index;
10.     end
11.     index = index + 1;
12. end
13. fprintf('the max value in A is %d at %d\n', ...
14.             theMax, theIndex);
```


Exercise 4.4 Producing `while` statement results

Enter and run the script in Listing 4.7. It should produce the following results:

```
A =
  6 12 6 91 13 61 26 22 71 54
the max value in A is 91 at 4
>>
```

4.7.3 Loop-and-a-Half Implementation

Listing 4.8 illustrates the implementation of the loop-and-a-half iteration style, in which we must enter the loop and perform some computation before realizing that we do not need to continue. Here we continually ask the user for the radius of a circle until an illegal radius is entered, which is our cue to terminate the iteration. For each radius entered, we want to display the area and circumference of the circle with that radius.

Style Points 4.3

We wrote the `for` loop examples in two styles: the direct access style and the indexing style. Many people code in the indexing style even when the index value is not explicitly required. This is slightly tacky and demonstrates a lack of appreciation for the full power of the MATLAB language.

Style Points 4.4

The use of `break` and `continue` statements is frowned upon in programming circles for the same reason that the `goto` statement has fallen into disrepute—they make it more difficult to understand the flow of control through a complex program. It is preferable to express the logic for remaining in a `while` loop explicitly in its controlling logical expression, combined with `if` statements inside the loop to skip blocks of code. However, sometimes this latter approach causes code to be more complex than would be the case with judicious use of `break` or `continue`.

In Listing 4.8:

Line 1: Initializes the radius value to allow the loop to be entered the first time.

Line 2: We will remain in this loop until the user enters an illegal radius.

Line 3: The `input(...)function` shows the user the text string, parses what is typed, and stores the result in the variable provided. This is described fully in Chapter 8.

Line 4: We want to present the area and circumference only if the radius has a legal value. Since this test occurs in the middle of the `while` loop, we call this "loop-and-a-half" processing.

Lines 5–8: Compute and display the area and circumference of a circle.

Try this script in Exercise 4.5.

Listing 4.8 Loop-and-a-half example

```
 1. clear
 2. clc
 3. close all
    % Listing 04.08 Loop-and-a-half example
 4. R = 1;
 5. while R > 0
 6.     R = input('Enter a radius: ');
 7.     if R > 0
 8.         area = pi * R^2;
 9.         circum = 2 * pi * R;
10.         fprintf('area = %f; circum = %f\n', ...
11.             area, circum);
12.     end
13. end
```

Exercise 4.5 Producing loop-and-a-half test results

Enter and run the script in Listing 4.8. It should produce the following results:

```
Enter a radius: 4
area = 50.265482; circum = 25.132741
Enter a radius: 3
area = 28.274334; circum = 18.849556
Enter a radius: 100
area = 31415.926536; circum = 628.318531
Enter a radius: 0
>>
```

4.7.4 Breaking a `while` Loop

As with the `for` loop, `break` will exit the innermost `while` loop, and `continue` will skip to the end of the loop but remain within it.

4.8 Engineering Example—Computing Liquid Levels

Figure 4.5 shows a cylindrical tank of height *H* and radius *r* with a spherical cap on each end (also of radius, *r*). If the height of the liquid is *h*, what is the volume of liquid in the tank? Clearly, the calculation of the volume of liquid in the tank depends upon the relationship between *h, H,* and *r*:

- If *h* is less than *r*, we need the volume, *v*, of a partially filled sphere given by:

$$v = \frac{1}{3}\pi h^2(3r - h)$$

- If *h* is greater than *r* but less than $H - r$, we need the volume of a fully filled hemisphere plus the volume of a cylinder of height $h - r$:

$$v = \frac{2}{3}\pi r^3 + \pi r^2(h - r)$$

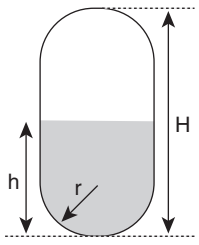

Figure 4.5 *A tank containing liquid*

- If h is greater than $H + r$, we need the volume of a fully filled sphere plus the volume of a cylinder of height $H + 2r$ minus the partially empty upper hemisphere of height $H + h$:

$$v = \frac{4}{3}\pi r^3 + \pi r^2(H - 2r) - \frac{1}{3}\pi(H - h)^2(3r - H + h)$$

The script to perform this calculation is shown in Listing 4.9. Rather than performing the computations for one liquid level only, we should write the script so that we continue to consider tanks of different dimensions and different liquid heights for each tank until the user indicates that he needs no more results.

In Listing 4.9:

Line 1: Initializes the value to keep it in the first `while` loop.

Lines 3 and 4: Get the tank sizes.

Line 5: Initializes the value to keep it in the inner `while` loop.

Line 7: Gets the liquid height.

Lines 8–14: Calculations for legal values of h. Notice that no dot operators are required here, because these conditional computations will not work correctly with vectors of H, r, or h.

Listing 4.9 Script to compute liquid levels

```
 1. another_tank = true;
 2. while another_tank
 3.     H = input('Overall tank height: ');
 4.     r = input('tank radius: ');
 5.     more_heights = true;
 6.     while more_heights
 7.         h = input('liquid height: ');
 8.         if h < r
 9.            v = (1/3)*pi*h.^2.*(3*r-h);
10.         elseif h < H-r
11.             v = (2/3)*pi*r^3 + pi*r^2*(h-r);
12.         elseif h <= H
13.             v = (4/3)*pi*r^3 + pi*r^2*(H-2*r) ...
14.                 - (1/3)*pi*(H-h)^2*(3*r-H+h);
15.         else
16.             disp('liquid level too high')
17.             continue
18.         end
19.         fprintf( ...
20.         'rad %0.2f ht %0.2f level %0.2f vol %0.2f\n', ...
21.                   r,          H,             h,        v);
22.         more_heights = input('more levels? (y/n)','s')=='y';
23.    end
24.    another_tank = input('another tank? (y/n)','s')=='y';
25. end
```

Table 4.1 Results for liquid levels

```
Overall tank height: 10
tank radius: 2
liquid height: 1
radius 2.00 height 10.00 level 1.00 vol 5.24
more levels? (y/n)y
liquid height: 8
radius 4.00 height 8.00 level 8.00 vol 268.08
more levels? (y/n)
another tank? (y/n)
```

Lines 15 and 16: Illegal *h* values end up here.
Line 17: Goes to the end of the inner loop, skipping the printout.
Lines 19–21: Print the result.
Line 22: More levels when "y" is entered.
Line 24: Another tank when "y" is entered.

Table 4.1 shows some typical results.

Chapter Summary

This chapter presented techniques for changing the flow of control of a program for condition execution and repetitive execution:

- The most general conditional form is the `if` statement, with or without the accompanying `elseif` and `else` statements
- The `switch` statement considers different cases of the values of a countable variable
- A `for` loop in its most basic form executes a code block for each of the elements of a vector
- A `while` loop repeats a code block a variable number of times, as long as the conditions specified for continuing the repetition remain true

Special Characters, Reserved Words, and Functions

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`false`	Logical false	4.2
`true`	Logical true	4.2
`break`	A command within a loop module that forces control to the statement following the innermost loop	4.6.4, 4.7.4
`case`	A specific value within a `switch` statement	4.1, 4.4.1
`continue`	Skips to the end of the innermost loop, but remains inside it	4.6.4, 4.7.4
`else`	Within an `if` statement, begins the code block executed when the condition is false	4.1, 4.3.2
`elseif`	Within an `if` statement, begins a second test when the first condition is false	4.1, 4.3.2
`end`	Terminates an `if`, `switch`, `for`, or `while` module	4.1, 4.3.2
`for` var = v	A code module repeats as many times as there are elements in the vector `v`	4.1, 4.6
`if <exp>`	Begins a conditional module; the following code block is executed if the logical expression `<exp>` is true	4.1, 4.3.2
`input(str)`	Requests and parses input from the user	4.3.2
`otherwise`	Catch-all code block at the end of a `switch` statement	4.1, 4.4.1
`switch(variable)`	Begins a code module selecting specific values of the `variable` (must be countable)	4.1, 4.4.1
`while <exp>`	A code module repeats as long as the logical expression `<exp>` is true	4.1
`all(a)`	True if all the values in `a`, a logical vector, a, are true	4.3.3
`and(a, b)`	True if both `a` and `b` are true (can be vectors)	4.3.3
`any(a)`	True if any of the values in `a`, a logical vector, is true	4.3.3
`not(a)`	True if `a` is false; false if `a` is true (can be vectors)	4.3.3
`or(a, b)`	True if either `a` or `b` is true (can be vectors)	4.3.3

Self Test

Use the following questions to check your understanding of the material in this chapter:

True or False

1. MATLAB keywords are colored green by the editor.
2. Indentation is required in MATLAB to define code blocks.
3. It is possible that no code at all is executed by `if` or `switch` constructs.
4. The word `true` is a valid logical expression.
5. When evaluating a sequence of logical `&&` expressions, MATLAB will stop processing when it finds the first `true` result.
6. The `for` loop repeats the enclosed code block a fixed number of times even if you modify the index variable within the code block.
7. Using a `break` statement is illegal in a `while` loop.
8. The logical expression used in a `while` loop specifies the conditions for exiting the loop.

Fill in the Blanks

1. MATLAB uses _____ in the text to define the extent of code blocks.
2. The function _____is implicitly called by MATLAB if you supply a vector of logical values to the `if` statement.
3. It is good practice to include _____ in a `switch` statement to trap illegal values entering the `switch`.
4. There is no reason to evaluate any more components of a logical or expression once a(n) _____ result has been found.
5. A `while` loop can be repeated a number of times, depending on the _____ being processed.
6. If you are in a(n) _____ loop, you can use the `break` statement to skip immediately out of the _____ loop.

Programming Projects

1. Write a script to solve this problem. Assume that you have a vector named D. Using iteration (`for` and/or `while`) and conditionals (`if` and/or `switch`), separate vector D into four vectors `posEven`, `negEven`, `posOdd`, and `negOdd`.
   - `posEven` contains all of the positive even numbers in D.
   - `negEven` contains all of the negative even numbers in D.
   - `posOdd` contains all of the positive odd numbers in D.
   - `negOdd` contains all of the negative odd numbers in D.

For example:

```
if D = [-4,-3,-2,-1,0,1,2,3,4],
posEven=[2,4], negEven=[-4,-2],
posOdd=[1,3] and negOdd=[-3,-1]
```

2. You must use either `for` or `while` to solve the following problems.
   a. Iterate through a vector, `A`, using a `for` loop, and create a new vector, `B`, containing `logical` values. The new vector should contain `true` for positive values and `false` for all other values. For example, if `A = [-300 2 5 -63 4 0 -46]`, the result should be `B = [false true true false true true false]`
   b. Iterate through the vector, `A`, using a `while` loop, and return a new vector, `B`, containing `true` for positive values and `false` for all other values.
   c. Iterate through a logical array, `N`, using a `for` loop, and return a new vector, `M`, containing the value 2 wherever an element of `N` is `true` and the value −1 (not a logical value) wherever `N` is `false`. For example, if `N = [true false false true true false true]`, the result should be `M = [2 -1 -1 2 2 -1 2]`
   d. Iterate through an array, `Z`, using a `while` loop. Replace every element with the number 3 until you reach a number larger than 50. Leave the rest unchanged. For example, if `Z = [4 3 2 5 7 9 0 64 34 43]`, after running your script, `Z = [3 3 3 3 3 3 3 3 34 43]`

3. You are hiring grad students to work for your company, which you have recently started. The Human Resources department has asked you to write a script that will help them determine the chances of an individual applicant getting a job after interviewing. The following table outlines the rules for determining the chances for the applicant to get a job:

GPA Value	Chance of Being Hired
GPA>= 3.5	90%
3.0<= GPA < 3.5	80%
2.5 <= GPA < 3.0	70%
2.0 <= GPA < 2.5	60%
1.5 <= GPA < 2.0	40%
GPA < 1.5	30%

Your script should repeatedly ask the user for a GPA value and compute the student's chances of being hired. It should continue asking for GPA values until a negative number is entered. For example:

- GPA input: 4 should give the answer 0.9
- GPA input: 3.5 should give the answer 0.9
- GPA input: 3.4 should give the answer 0.8

4. You were just hired for a summer internship with one of the area's best software companies; however, on your first day of work you learn that for the next three months, the only job you will have is to convert binary (base 2) numbers into decimal numbers (base 10). You decide to write a script that will repetitively ask the user for a binary number and return its decimal equivalent until an illegal number (one containing digits other than 0 or 1) is entered. The number entered should contain only the digits 0 and 1. The rightmost digit has the value 2^0 and the digit N places to the left of that has the value 2^N. For example, entering `110101` returns

$$53 = 2^5 + 2^4 + 2^3 + 2^0$$

You must use iteration to solve this problem. Note: The `input (...)` function prompts the user for a value, parses the characters entered according to normal MATLAB rules, and returns the result.

5. You have a friend who has too many clothes to store in his or her tiny wardrobe. Being a good friend, you offer to help to decide whether each piece of clothing is worth saving. You decide to write a script that will compute the value of each piece of clothing. A piece of clothing has five attributes that can be used to determine its value. The attributes are: condition, color, price, number of matches, and comfort. Each attribute will be rated on a scale of 1 to 5. Write a script called `clothes` that will ask the user for the ratings for each attribute and store the result in a vector. The order of attributes in the vector is: `[condition color price matches comfort]`

   The script should compute a value between 0 and 100; 100 represents a good piece of clothing, while 0 represents a bad piece of clothing. The points that should be given for each attribute are shown below:

Condition:	1=>0; 2=>5; 3=>10; 4=>15; 5=>20
Color:	1 => blue => 12; 2 => red => 2; 3 => pink => 15; 4 => yellow => 20; 5 => white => 12
Price:	1 => 8, 2–3 => 16, 4–5 => 20
Matches:	1–2 => 8, 3–5 => 19
Comfort:	1 => 6, 2–3 => 13, 4–5 =>18

   Note: If a number other than 1–5 is assigned for one of the attributes, no points should be given.

6. A "yard" is a traditional English container. It is 36 inches long, and can be approximated by a 4-inch diameter glass sphere attached to a conical section whose narrow end is 1 inch in diameter, and

whose wide end is 6 inches in diameter. Write a script to do the following:

a. ask the user for the height of the liquid in the yard, and
b. calculate the volume of liquid needed to fill the yard to that level.

7. Now that you're comfortable with iteration, you're going to have to solve an interesting problem. It seems that the Math department at a rival university has once again dropped the ball, and forgotten the value of `pi`. You are to write a function called `mypi`, which consumes a number that specifies the required accuracy and then approximates the value of `pi` to that accuracy. You are going to use the following algorithm based on geometric probability.

   Think about a quarter circle inside of a unit square (the quarter circle has area $\pi/4$). You pick a random point inside the square. If it is in the quarter circle, you get a "hit"; and if not, you get a "miss." The approximate area of the quarter circle will be given by the number of hits divided by the number of points you chose.

   Hint

   you could use the function `rand(...)` in this problem.

   Your function should repeat the process of counting hits and misses until at least 10,000 tries have been made, and the successive estimates of `pi` are within the prescribed accuracy. It should return the estimated value of `pi`.