72 DPI M06_SMIT8708_03_SE_C06.indd Page 121 02/02/12 3:06 PM user-f396 F-396 EQA

CHAPTER 6

Character Strings

6.1 Character String Concepts: Mapping Casting, Tokens, and Delimiting
6.2 MATLAB Implementation
6.2.1 Slicing and Concatenating Strings
6.2.2 Arithmetic and Logical Operations
6.2.3 Useful Functions
6.3 Format Conversion Functions
6.3.1 Conversion from Numbers to Strings
6.3.2 Conversion from Strings to Numbers
6.4 Character String Operations
6.4.1 Simple Data Output: The `disp(...)` Function
6.4.2 Complex Output
6.4.3 Comparing Strings
6.5 Arrays of Strings
6.6 Engineering Example—Encryption

Chapter Objectives

This chapter discusses the nature, implementation, and behavior of character strings in the MATLAB language:

- The internal workings of character strings as vectors
- Operations on character strings
- Converting between numeric and character string representations
- Input and output functions
- The construction and uses for arrays of strings

To this point in the text, we have seen the use of character strings that we can store in variables and display in the Command window. In reality, we have already seen a significant amount of character manipulation that we have taken for granted. The m-files we use to store scripts and functions contain lines of legible characters separated by an invisible “new-line” character.

Introduction

This chapter presents the underlying concept of character storage and the tools MATLAB provides for operating on character strings. We need to distinguish two different relationships between characters and numbers:

1. Individual characters have an internal numerical representation: the visible character shapes we see in windows are created as a collection of white and black dots by special software called a **character generator**. Character generators allow us to take the underlying concept of a character—say, “w”—and “draw” that character on screen or paper in accordance with the rules defined by the current font. A complete study of fonts is beyond the scope of this discussion, but we need to understand how computers in general and the MATLAB language in particular represent that “underlying concept” of a

72 DPI M06_SMIT8708_03_SE_C06.indd Page 122 02/02/12 3:06 PM user-f396 F-396 EQA

character. This is achieved by representing each individual character by its numerical equivalent. Not long ago, there were many different representations. Today, the dominant representation is the one defined by the American Standard Code for Information Interchange (ASCII). In this representation, the most common uppercase and lowercase characters, numbers, and many punctuation marks are represented by numbers between 0 and 127. A complete listing of the first 255 values is included in Appendix B.

2. Strings of characters represent numerical values to the user: numerical values are stored in a special, internal representation for efficient numerical computation as described in Appendix C. However, whenever we need to see the value of that number in the Command window, that internal representation is automatically converted by MATLAB into a character string representing its value in a form we can read. For example, if the variable `a` contained the integer value 124, internally that number could be stored in a single byte (8 bits) with a binary value of 011111100—not a very meaningful representation, but efficient internally for performing arithmetic and logical operations. For the user to understand that value, internal MATLAB logic must convert it to the three printable characters: `'124'`. Similarly, when we type in the Command window or use the `input(...)` function, the set of characters that we enter is automatically translated from a character string into the internal number representation.

6.1 Character String Concepts: Mapping Casting, Tokens, and Delimiting

Here we see the MATLAB language tools that deal with the first relationship between characters and numbers: the numerical representation of individual characters.

The basic idea of **mapping** is that it defines a relationship between two entities. The most obvious example of mapping is the idea that the function $f(x) = x^2$ defines the mapping between the value of x and the value of $f(x)$. We will apply that concept to the process of translating a character (like "A") from its graphical form to a numerical internal code. **Character mapping** allows each individual graphic character to be uniquely represented by a numerical value.

Casting is the process of changing the way a language views a piece of data without actually changing the data value. Under normal circumstances, a language like MATLAB automatically presents a set of data in the "right" form. However, there are times when we wish to force the language to treat a data item in a specific way. For example, if we create a variable

72 DPI M06_SMIT8708_03_SE_C06.indd Page 123 02/02/12 3:06 PM user-f396 F-396 EQA

containing a character string, MATLAB will consistently display it as a character string. However, we might want to view the underlying numerical representation as a number, in which case we have to cast the variable containing the characters to a numerical data type. MATLAB implements casting as a function with the name of the data type expected. In essence, these functions implement the mapping from one character representation to another.

A **token** is a collection of characters to which we may wish to attach meaning. Obvious examples of tokens are the name of a MATLAB variable or the characters representing the values of a number to be used in an expression.

A **delimiter** is a character used to separate tokens. The space character, for example, can delimit words in a sentence; punctuation marks provide additional delimiters with specific meanings.

6.2 MATLAB Implementation

When you enter a string in the Command window or the editor, MATLAB requires that you delimit the characters of a string with a single quote mark (`'`). Note that you can include a single quote mark within the string by doubling the character. For example, if you entered the following in the Command window:

```
>>refusal = 'I can''t do that!'
```

The result displayed would be

```
refusal = I can't do that
```

Exercise 6.1 illustrates the concept of casting between data types `char` and `double`.

In Exercise 6.1 the casting function `uint8(...)` takes a character or character string and changes its representation to a vector of the same length as the original string. Then the casting function `char(...)` takes a number or vector and causes it to be presented as a string. The casting function `double(...)` appears to act in the same way as `uint8(...)`, but it actually uses 64 bits to store the values. Single quotes delimit a string to be assigned to the variable `fred`. Notice that when a string is presented as a result, the delimiters are omitted. When you apply arithmetic operations to a string, the operation is illegal on characters; therefore, an implicit casting to the numerical equivalent occurs.

You can perform any mathematical operation on the vector and use the cast, `char(...)`, to cast it back to a string.

72 DPI M06_SMIT8708_03_SE_C06.indd Page 124 02/02/12 3:06 PM user-f396 F-396 EQA

Exercise 6.1 Character casting

Enter the following in the Command window and study the results:

```
>> uint8('A') % uint8 is an integer data type
              % with values 0 - 255
ans =
   65
>> char(100) % char is the character class
ans =
d
>> char([97 98 99 100 101])
ans =
abcde
>> double('fred')
ans =
   102 114 101 100
>> fred = 'Fred'
fred =
Fred
>> next = fred + 1
next =
   71 115 102 101
>> a = uint8(fred)
a =
   70 114 101 100
>> name = char(a + 1)
name =
Gsfe
```

6.2.1 Slicing and Concatenating Strings

Strings are internally represented as vectors; therefore, we can perform all the usual vector operations on strings. Try it in Exercise 6.2.

Exercise 6.2 Character strings

```
>> first = 'Fred'
first =
Fred
>> last = 'Jones'
last =
Jones
>> name = [first, ' ', last]
name =
Fred Jones
>> name(1:2:end)
ans =
Fe oe
>> name(end:-1:1)
ans =
senoJ derF
```

72 DPI M06_SMIT8708_03_SE_C06.indd Page 125 03/02/12 7:20 PM user-f396 F-396 EQA

Exercise 6.3 Character string logic

```
>> n = 'fred'
n =
fred
>> n > 'g'
ans =
0  1  0  0
```

6.2.2 Arithmetic and Logical Operations

Mathematical operations can be performed on the numerical mapping of a character string. If you do not explicitly perform that casting first, MATLAB will do the cast for you and create a result of type double (not usually suitable for character values). Note that `char('a' + 1)` returning `'b'` is an accident of the character type mapping.

Logical operations on character strings are also exactly equivalent to logical operations on vectors, with the same automatic casting. Exercise 6.3 gives you an opportunity to try it yourself.

6.2.3 Useful Functions

The following functions are useful in analyzing character strings:

- `ischar(a)` returns `true` if `a` is a character string
- `isspace(ch)` returns `true` if the character `ch` is the space character

6.3 Format Conversion Functions

Now we turn to the second relationship between characters and numbers: using character strings to represent individual number values. We need two separate capabilities: converting numbers from the efficient, internal form to legible strings and converting strings provided by users of MATLAB into the internal number representation. MATLAB provides a number of functions that transform data between string format and numerical format.

6.3.1 Conversion from Numbers to Strings

Use the following built-in MATLAB functions for a simple conversion of a single number, `x`, to its string representation:

- `int2str(x)` if you want it displayed as an integer value
- `num2str(x, n)` to see the decimal parts; the parameter `n` represents the number of decimal places required—if not specified, its default value is 3

72 DPI M06_SMIT8708_03_SE_C06.indd Page 126 02/02/12 3:06 PM user-f396 F-396 EQA

Frequently you need better control over the data conversion, and the function `sprintf(...)` provides fine-grained control. The MATLAB version of `sprintf(...)` is very similar to the C / C++ implementation of this capability. The first parameter to `sprintf` is a **format control string** that defines exactly how the resulting string should be formatted. A variable number of **value parameters** follow the format string, providing data items as necessary to satisfy the formatting.

Basically the format string contains characters to be copied to the result string; however, it also contains two types of special entry introduced by the following two special characters:

- The `'%'` character introduces a conversion specification, indicating how one of the value parameters should be represented. The most common conversions are `%d` (integer), `%f` (real), `%g` (general), `%c` (character), and `%s` (string). A number may be placed immediately after the `%` character to specify the minimum number of characters in the conversion. If more characters than the specified minimum are required to represent the data, they will be added. In addition, the `%f` and `%g` conversions can include `'.n'` to indicate the number of decimal places required. If you actually want a `'%'` character, it must be doubled, for example, `'%%'`. MATLAB processes each of the value parameters in turn, inserting them in the result string according to the corresponding conversion specification. If there are more parameters than conversion specifications in the format control string, the format control string is repeated.
- The `'\'` character introduces format control information, the most common of which are `\n` (new line) and `\t` (tab). If the `'\'` character is actually wanted in the result string, it should be doubled, for example, `'\\'`.

Consider the following statements:

```
A = [4.7 1321454.47 4.8];
index = 1;
v = 'values';
str = sprintf('%8s of A(%d) are \t%8.3f\t%12.4g\t%f\n'...
    v, index, A(index,1), A(index,2), A(index,3))
str =
      values of A(1) are  4.700       1.321e+006       4.800000
```

The first conversion, `'%8s'`, took the value of the first parameter, `v`, allowed eight spaces for its conversion, and copied its contents to the result. Since this was a string conversion, the characters were merely copied. The characters `' of A('` were then appended to the output string. The second conversion, `'%d'`, took the value of the second parameter, `index`, and converted it as an integer with the minimum space allocated. The characters `') are'` were then appended to the output string, followed by a tab character that inserted

72 DPI M06_SMIT8708_03_SE_C06.indd Page 127 02/02/12 3:06 PM user-f396 F-396 EQA

enough spaces to bring the next characters to a column that is an even multiple of eight. The following three conversions appended the next three value parameters converted with three decimal places, a general conversion with at least 12 spaces and 4 decimal places, and the default numerical conversion. Finally, a new line character was inserted into the string.

6.3.2 Conversion from Strings to Numbers

Conversion from strings to numbers is much messier, and it should be avoided if possible. When possible, allow MATLAB's built-in function `input(...)` to do the conversion for you. If you have to do the conversion yourself, you can either split a string into tokens and then convert each token with the `str2num(str)` function or, if you are really desperate and using licensed MATLAB software, you can use the function `sscanf(...)`.

The function `input(str)` presents the string parameter to the user in the Command window and waits for the user to type some characters and the Enter key, all of which are echoed in the Command window. Then it parses the input string according to the following rules:

- If the string begins with a numerical character, MATLAB converts the string to a number
- If it begins with a non-numeric character, MATLAB constructs a variable name and looks for its current value
- If it begins with an open bracket, `'['`, a vector is constructed
- If it begins with the single quote character, MATLAB creates a string
- If a format error occurs, MATLAB repeats the prompt

This behavior can be modified if `'s'` is provided as the second parameter, `input(str, 's')`, in which case the complete input character sequence is saved as a string. Exercise 6.4 demonstrates a number of capabilities of the `input(...)` function.

In Exercise 6.4, first we define the variable `fred`. Then MATLAB attempts to interpret the result either as a number or as the name of an existing variable. Since the variable `fred` was defined (although not a number), it was assigned correctly to the variable `n`. MATLAB will distinguish between a variable and a number input by the first digit. Here, the information entered was an illegal variable name beginning with a number. When `input(...)` detects an error parsing the text entered, it automatically resets and requests a new entry.

On the second attempt, although this is a correctly formed variable name, its value is not known. On the third attempt, the `input(...)` function actually treats the string entered as an expression, to be evaluated by the same process as MATLAB parses the Command window entries.

72 DPI M06_SMIT8708_03_SE_C06.indd Page 128 02/02/12 3:06 PM user-f396 F-396 EQA

Exercise 6.4 The `input(...)` function

```
>> fred = 'Fred';
>> n = input('Enter a number: ')
Enter a number: 5
n =
     5
>> n = input('Enter a number: ')
Enter a number: fred
n =
Fred
>> n = input('Enter a number: ')
Enter a number: 1sdf
??? 1sdf
Error: Missing MATLAB operator.
Enter a number: s1df
??? Error using ==> input
Undefined function or variable 's1df'.
Enter a number: char(fred - 2)
n =
Dpcb
>> n = input('Enter a number: ')
Enter a number: 'ABCD'
n =
ABCD
>> n = input('Enter a number: ', 's' )
Enter a number: ABCD
n =
ABCD
```

If you actually want a string literal entered, it must be enclosed in the string delimiters. If you are sure you want a string literal entered, the second parameter, `'s'`, forces MATLAB to return the string entered without attempting to parse it.

The function `str2num(str)` consumes a token (string) representing a single numerical value and returns the numerical equivalent. Do Exercise 6.5 to understand this function.

Exercise 6.5 Converting strings to numbers

```
>> value = str2num('3.14159')
value =
       3.1416
Now, to check the class of the variable value, either look in the
Variables window or enter the whos command:
>> whos
    Name        Size           Bytes Class         Attributes
    value       1x1                8 double
>>
```

We observe that the function has indeed interpreted the string as its numerical value.

72 DPI M06_SMIT8708_03_SE_C06.indd Page 129 02/02/12 3:06 PM user-f396 F-396 EQA

The function `sscanf(...)` was designed to extract the values of variables from a string, but is really difficult to use. The author recommends the use of `strtok(...)` followed by `str2num(...)` as necessary to accomplish the same goal in a more controlled manner.

6.4 Character String Operations

As with the string-to-number conversions, input and output in the Command window can be accomplished with simple functions that have little flexibility or with complex functions that have better control.

6.4.1 Simple Data Output: The `disp(. . .)` Function

We have already seen the use of the `disp(...)` function to present data in readable form in the Intractions window. As the exercises indicate, it can present the values of any variable, regardless of type, or of strings constructed by concatenation. Note, however, that an explicit number conversion is required to concatenate variables with strings. Try Exercise 6.6.

Note that although you can concatenate strings for output, conversion from the ASCII code is not automatic; the second result produced a character whose ASCII code is 4. You must use the simple string conversion functions to enforce consistent information for concatenation.

6.4.2 Complex Output

The function `fprintf(...)` is similar to `sprintf(...)`, except that it prints its results to the Command window instead of returning a string. `fprintf(...)` returns the number of characters actually printed. Exercise 6.7 demonstrates this.

6.4.3 Comparing Strings

Since strings are readily translated into vectors of numbers, they may be compared in the obvious way with the logical operators we used on numbers. However, there is the restriction that either the strings must be

Exercise 6.6 The `disp(...)` function

```
>> a = 4;
>> disp(a)
     4
>> disp(['the answer is ', a])
the answer is
>> disp(['the answer is ', int2str(a)])
the answer is 4
```

72 DPI M06_SMIT8708_03_SE_C06.indd Page 130 03/02/12 7:21 PM user-f396 F-396 EQA

Exercise 6.7 `fprintf(...)` and `sprintf(. . .)`

```
>> a = 42;
>> b = 'fried okra';
>> n = fprintf('the answer is %d\n cooking %s', ...
                                 a,            b);
the answer is 42
cooking fried okra
n =
   37
>> s = sprintf('the answer is %d\n cooking %s\n', ...
                                 a,           b)
s =
the answer is 42
 cooking fried Okra
>> str = input('Enter the data: ', 's');
Enter the data: 42 3.14159 -1
A = sscanf( str,'%f')
A =
   42.0000
    3.1416
   -1.0000
>>
```

of the same length or one of them must be of length 1 before it is legal to compare them with these operators. To avoid this restriction, MATLAB provides the C-style function `strcmp(<s1>, <s2>)` that returns `true` if the strings are identical and `false` if they are not.

Unfortunately, this is not quite the same behavior as the C version, which does a more rigorous comparison returning −1, 0, or 1. You can try a character string comparison in Exercise 6.8.

Exercise 6.8 Character string comparison

```
>> 'abcd' == 'abcd'
    1     1     1     1
>> 'abcd' == 'abcde'
??? Error using ==> eq
Array dimensions must match for binary array op.
>> strcmp('abcd', 'abcde')
ans =
     0
>> strcmp('abcd', 'abcd')
ans =
     1
>> 'abc' == 'a'
ans =
     1     0     0
>> strcmpi('ABcd', 'abcd')
ans =
     1
```

72 DPI M06_SMIT8708_03_SE_C06.indd Page 131 02/02/12 3:06 PM user-f396 F-396 EQA

Common Pitfalls 6.1

The `if` statement uses a logical expression as its controlling test; therefore, it is bound by the same comparison rules as those applied to vectors. Two strings being compared must be of the same length, and all of the comparisons must match to result in a logical `true`. Frequently, we expect the `if` statement to compare strings of unequal length. However, this will cause an error whenever two strings of unequal length are compared (unless one string is just one character). You should use the `switch` statement, which will correctly compare strings of unequal length in the case tests.

In Exercise 6.8, we see that strings of the same length compare exactly to vectors returning a logical vector result. You cannot use the equality test on strings of unequal length. `strcmp(...)` deals gracefully with strings of unequal length. As with vectors, the equality test works if one of the inputs is a single character. For case-independent testing, use `strcmpi(...)`.

6.5 Arrays of Strings

Since a single character string is stored as a vector, it seems natural to consider storing a collection of strings as an array. The most obvious way to do this, as shown in previous examples, has some limitations, for which there are nice, tidy cures built into the MATLAB language. Consider the example shown in Exercise 6.9. Character arrays can be constructed by either of the following:

- As a vertical vector of strings, all of which must be the same length

Exercise 6.9 Character string arrays

```
>> v = ['Character strings having more than'
        'one row must have the same number '
        'of columns just like arrays!      ']
v =
Character strings having more than
one row must have the same number
of columns just like arrays!
>> v = ['MATLAB gets upset'
        'when rows have'
        'different lengths']
??? Error using ==> vertcat
All rows in the bracketed expression must have the
same number of columns.

>>eng=char('Timoshenko','Maxwell','Mach','von Braun')
eng =
Timoshenko
Maxwell
Mach
von Braun
>> size(eng)
ans =
     4    10
```

72 DPI M06_SMIT8708_03_SE_C06.indd Page 132 02/02/12 3:06 PM user-f396 F-396 EQA

Common Pitfalls 6.2

Trying to concatenate strings of unequal length vertically into column arrays of strings will cause errors because the vertical concatenation must use rows of equal length. Use the version of the `char(...)` function that pads the strings with spaces.

- By using a special version of the `char( &)` cast function that accepts a variable number of strings with different lengths, pads them with blanks to make all rows the same length, and stores them in an array of characters

6.6 Engineering Example—Encryption

The Problem

As public access to information becomes more pervasive, there is increasing interest in the use of encryption to protect intellectual property and private communications from unauthorized access. The following discussion is based on no direct knowledge of the latest encryption technology. However, it illustrates a very simple approach to developing an algorithm that is immune to all but the most obvious, brute-force code-breaking techniques.

Background

Historically, simple encryption has been accomplished by substituting one character for another in the message, so that `'Fred'` becomes `'Iuhg'` when substituting the letter three places down the alphabet for each letter in the message. More advanced techniques use a random letter selection to substitute new letters. However, any constant letter substitution is vulnerable to elementary code-cracking techniques based on the frequency of letters in the alphabet, for example.

The Solution

We propose a simple algorithm where a predetermined random series is used to select the replacement letters. Since the same letter in the original message is never replaced by the same substitute, no simple language analysis will crack the code. The `rand(...)` function is an excellent source for an appropriate random sequence. If the encryption and decryption processes use the same value to seed the generator, the same sequence of apparently random **(pseudo-random)** values will be generated.

Since the seed can take on 2^{31}–2 values, it is virtually impossible to determine the decryption without knowing the seed value. The seed (i.e., the decryption key) can be transmitted to anyone authorized to decrypt the message by any number of ways. Furthermore, since there are abundant different techniques for generating pseudo-random sequences, the specific generation technique must be known in addition to the seed value for successful decryption. Listing 6.1 shows the code for encrypting and

72 DPI M06_SMIT8708_03_SE_C06.indd Page 133 02/02/12 3:06 PM user-f396 F-396 EQA

Listing 6.1 Encryption exercise

```
disp('original text')
txt = ['For example, consider the following:' 13 ...
  'A = [4.7 1321454.47 4.8];' 13 ...
  'index = 1;' 13 ...
  'v = ''values'';' 13 ...
  'str = sprintf(''%8s of A(%d) are \t%8.3f ' 13 ...
  ' v, index, A(index,1) ' 13 ...
  'str = ' 13 ...
  ' values of A(1) are 4.700' 13 ...
  'The first conversion, ''%8s'', took the value' ...
  ' of the first ' ...
  'parameter, v, allowed 8 spaces. ' 13 ]
  % % encryption section
rand('state', 123456)
loch = 33;
hich = 126;
range = hich+1-loch;
rn = floor( range * rand(1, length(txt) ) );
change = (txt>=loch) & (txt<=hich);
enc = txt;
enc(change) = enc(change) + rn(change);
enc(enc > hich) = enc(enc > hich) - range;
disp('encrypted text')
encrypt = char(enc)
% % good decryption
rand('state', 123456);
rn = floor( range * rand(1, length(txt) ) );
change = (encrypt>=loch) & (encrypt<=hich);
dec = encrypt;
dec(change) = dec(change) - rn(change) + range;
dec(dec > hich) = dec(dec > hich) - range;
disp('good decrypt');
decrypt = char(dec)
% % bad seed
rand('seed', 123457);
rn = floor( range * rand(1, length(txt) ) );
change = (encrypt>=loch) & (encrypt<=hich);
dec = encrypt;
dec(change) = dec(change) - rn(change) + range;
dec(dec > hich) = dec(dec > hich) - range;
disp('decrypt with bad seed')
decrypt = char(dec)
% % different generator
rand('seed', 123456)
rn = mod(floor( range * abs(randn(1, length(txt) ))/10 ),  ...
     range);
change = (encrypt>=loch) & (encrypt<=hich);
dec = encrypt;
dec(change) = dec(change) - rn(change) + range;
dec(dec > hich) = dec(dec > hich) - range;
disp('decrypt with wrong generator')
decrypt = char(dec)
```

72 DPI M06_SMIT8708_03_SE_C06.indd Page 134 02/02/12 3:06 PM user-f396 F-396 EQA

decrypting by this technique and two attempts to decrypt—once with the wrong key and once with the wrong generator.

In Listing 6.1:

Lines 2–12: This is the original text taken from earlier in this chapter. Multiple lines of characters can be concatenated as shown. The number 13 inserted in the string is the numerical equivalent of the new line escape sequence, `'\n'`.

Line 13: Seeds the random generator with a known value.

Lines 14–16: Set the upper and lower bounds and the range of the characters we will convert. This range excludes 32, the space character, and 13, the new line character. This choice was deliberate—it leaves the encrypted text with the appearance of a character substitution algorithm since all the characters are printable, and seem to be grouped in words.

Line 17: Generates the random values between 0 and `range-1`.

Line 18: Identifies the indices of the printable characters.

Line 19: Makes a copy of the original text.

Line 20: Adds the random offsets to those characters we intend to change.

Line 21: If the addition pushes a character value above the maximum printable character, this brings it back within range.

Lines 22–23: Display the encrypted text. Notice that no two characters of the original text are replaced by the same character.

Lines 24–27: Begin the decryption by seeding the generator with the same value, creating the same random sequence, finding the printable characters, and copying the original file to the decrypt string.

Lines 28–29: We must subtract the random sequence from the encrypted string and correct for the underflow. However, there are some numerical issues involved. It is best to add the `range` value to all the letters while subtracting the random offsets, and then bring back those values that remain above the highest printable character.

Lines 30–31: Display the decrypted values.

Lines 32–39: Attempt to decrypt with the same code but a bad seed.

Lines 40–48: Attempt to decrypt with the right seed but a different generator—in this case, MATLAB's normal random generator limited to positive values within the letter range of interest.

Table 6.1 shows the output from this encryption exercise.

72 DPI M06_SMIT8708_03_SE_C06.indd Page 135 02/02/12 3:06 PM user-f396 F-396 EQA

Table 6.1 Encryption exercise results

`original text`
`txt =`
`For example, consider the following:`
`A = [4.7 1321454.47 4.8];`
`encrypted text encrypt =`
`@;J _a,Q/V_Q X/|IW?*q %;{ $Ctr:$&r3>`
`5 - v$zh uvqzmE@P(N Bh}.H`
`good decrypt`
`decrypt =`
`For example, consider the following:`
`A = [4.7 1321454.47 4.8];`
`decrypt with bad seed`
`decrypt =`
`tDQ <6VfMiS^ }1FI92/P c'@ eYrW%Q^2t+`
`6 L 4x5> B$rQ4XHpG# G;*<r`
`decrypt with wrong generator`
`decrypt =`
`>1E o-P:'P=p :xLjV+bi {!d 3)[Az$~c7<`
`' l fny& tHWB Vve6o`

Chapter Summary

This chapter discussed the nature, implementation, and behavior of character strings. We learned the following:

- Character strings are merely vectors of numbers that are presented to the user as single characters
- We can perform on strings the same operations that can be performed on vectors; if mathematical operations are performed, MATLAB first converts the characters to double values
- We can convert between string representations of numbers and the numbers themselves using built-in functions
- MATLAB provides functions that convert numbers to text strings for presentation in the Command window
- Arrays of strings can be assembled using the `char(...)` function

72 DPI M06_SMIT8708_03_SE_C06.indd Page 136 02/02/12 3:06 PM user-f396 F-396 EQA

Special Characters, Reserved Words, and Functions

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`'...'`	Encloses a literal character string	6.2
`char(...)`	Casts to a character type	6.2, 6.5
`disp(...)`	Displays matrix or text	6.4.1
`double(a)`	Casts to type `double`	6.2
`fprintf(...)`	Prints formatted information	6.4.2
`input(...)`	Prompts the user to enter a value	6.3.2
`int2str(a)`	Converts an integer to its numerical representation	6.3.1
`ischar(ch)`	Determines whether the given object is of type `char`	6.2.3
`isspace(a)`	Tests for the space character	6.2.3
`num2str(a,n)`	Converts a number to its numerical representation with `n` decimal places	6.3.1
`sscanf(...)`	Formatted input conversion	6.3.2
`sprintf(...)`	Formats a string result	6.3.1
`str2num(...)`	Convert a string to its numerical equivalent	6.3.2
`strcmp(s1, s2)`	Compares two strings; returns `true` if equal	6.4.3
`strcmpi(s1, s2)`	Compares two strings without regard to case; returns `true` if equal	6.4.3
`uint8(...)`	Casts to unsigned integer type with 8 bits	6.2

Self Test

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

True or False

1. Casting changes the value of a piece of data.
2. The ASCII code maps individual characters to their internal numerical representation.
3. Because the single quote mark (`'`) delimits strings, you cannot use it within a string.
4. If you attempt mathematical operations on a character string, MATLAB will throw an error.

72 DPI M06_SMIT8708_03_SE_C06.indd Page 137 02/02/12 3:06 PM user-f396 F-396 EQA

5. The function `disp(...)` can display multiple values to the Command window.
6. The function `strcmp(...)` throws an error if the two strings are of unequal length, unless one of them is a single character.
7. The `switch` statement will correctly compare strings of unequal length in the `case` tests.

Fill in the Blanks

1. Numerical values are stored in MATLAB in _______________ for efficient numerical computation.
2. Most common __________________, _______________, and many _________________are represented in ASCII by the numbers ________________.
3. The function ____________ casts a string to a vector of the same length as the string containing the numerical mapping of _______________________.
4. The function `fprintf(...)` requires a(n) _____________ that defines exactly how the resulting string should be formatted and a variable number of ________________.
5. Since the _____________ statement tests a logical expression, it _____________ test strings of unequal length.
6. A special version of the cast function accepts __________________ strings with different lengths, _____________, and stores them in an array of characters.

Programming Projects

1. Solve the following introductory problems on strings.
   a. Write a function `dayName` that consumes a parameter, `day`, containing the numerical value of a day in the month of September 2008. Your function should return the name of that day as a string. For example:

```
dayName( 8 ) should return 'Monday'
```

   b. You are now given a variable named `days`, a vector that contains the numeric values of days in the month of September 2008. Write a script that will convert each numeric value in the vector `days` into a string named `daysOfWeek` with the day names separated by a comma and

Hint

You should probably be concatenating the day names and the delimiters.

72 DPI M06_SMIT8708_03_SE_C06.indd Page 138 02/02/12 3:06 PM user-f396 F-396 EQA

a space. For example, `if days = [8, 9, 10], daysOfWeek` should be `'Monday, Tuesday, Wednesday'`

Notice that there is no separator before the first day name or after the last one.

2. Consider the problem the MATLAB system has in parsing the string:

   ```
   'V=[1 2 3 4; 5,6, 7;8; 9 10]'
   ```

   Your task is to use `strtok` to parse this line and construct the array it represents. You will write a function `arrayParse` that consumes a string and returns two variables: a string that is the variable name and an array.
   a. Tokenize the string first using `'='` as the delimiter to isolate the variable name and the expression to be evaluated. Return the variable name to the user and save the rest of the line as the variable `str1` for further processing. You may assume that there are no spaces outside the characters `'[. . .]'`.
   b. Tokenize `str1` with `'['` and `']'` to remove the concatenation operators and save the first token as `str2`.
   c. Tokenize `str2` using `';'` as the delimiter. This will produce 0 or more strings that represent the rows of the array. Save each in the variable `rowString`. You may assume for now that the first row is the longest one.
   d. Using nested `while` loops, tokenize each `rowString` with `','` and `' '` as delimiters and use `str2num`(. . .) to extract the numerical value of each array entry. Save it as `rowEntry`.
   e. Concatenate the `rowEntry` elements horizontally to produce each row of the array. If the row is too short, pad it with zeros.
   f. Concatenate each row vertically to produce the resulting array and return that array to the caller.
   g. Test the function with cases like:

   ```
   empty=[]
   row=[1 2 3 4]
   diag=[0 0 0 1; 0 0 1; 0 1; 1]
   ```

3. Write a function called `DNAcomplement` that consumes a set of letters as a character array that forms a `DNA` sequence such as `'gattaca'`. The function will produce the complement of the sequence so that `a's` become `t's`, `g's` become `c's`, and vice versa. The string `'gattaca'` would therefore become `'ctaatgt'`. You may assume that all the letters in the sequence will be lowercase and that they will all be either `a, t, g, or c`.

   Note: You may be tempted to use iteration for this problem, but you don't need it.

4. The function `rot(s, n)` is a simple Caesar cipher encryption algorithm that replaces each English letter in places forward or

72 DPI M06_SMIT8708_03_SE_C06.indd Page 139 02/02/12 3:06 PM user-f396 F-396 EQA

backward along the alphabet in the strings. For example, the result of `rot('Baz!',3)` is `'Edc!'`. An encrypted string can be deciphered by simply performing the inverse rotation on it, that is, `rot('Edc!',3)`, which rotates each English letter in the strings three places to the left. Numbers, symbols, and non-letters are not transformed. Implement the following function:

```
function rotatedText=rot(text,n)
```

To assist you as you solve this problem, you could write several functions as local functions in the `rot.m` file: `isUppercaseLetter(letter)`, `getUppercaseLetter(n)`, `getLowercaseLetter(n)`, and `getPosition(letter)`. You may also wish to use the built-in functions `isletter` (. . .), `find` (. . .), and `mod` (. . .).

5. You have a big problem. In one of your CS courses, your professor decides that the only way you will pass the class is if you write a function to get him out of a mess. All the grades in his class have been accidentally stored into one long string of characters containing only the letters A, B, C, D, F, and Y.
   a. Your job is to write a function called `CrazyGrade` that will take in the string and flip the grades according to the following specifications:

      `A` becomes `F`
      `B` becomes `D`
      `C` remains unchanged
      `D` becomes `B`
      `F` becomes `A`
      `Y` becomes `W`

   Your function should take in a string and return an inverted string. You may assume that the string will only consist of valid letter grades. For example,

   ```
   CrazyGrade('BADDAD') should return 'DFBBFB'
   CrazyGrade('BAYBAY') should return 'DFWDFW'
   ```

   b. To make matters worse, he wants you to organize this modified grade set. Write a function called `GradeDist` to bunch together all the similar grades (put all the `A`'s next to each other, `B`'s next to each other, etc.) Then, calculate and return the professor's grade distribution. Your function should take in a string and return a string with all similar grades grouped together, along with an array containing percentage values from `A`'s all the way to `F`'s. For example, if there are 15% `A`'s, 16% `B`'s, 33% `C`'s, 16% `D`'s, 16% `F`'s, and 4% `W`'s, `GradeDist` should return `[15 16 33 16 16 4]`.

72 DPI M06_SMIT8708_03_SE_C06.indd Page 140 02/02/12 3:06 PM user-f396 F-396 EQA