CHAPTER 8

Data Structures

KEY TERMS

data structures
cell array
structures
fields
database
record
categorical arrays
tables
sorting
index vectors
cells
content indexing
cell indexing
comma-separated list
dot operator
dynamic field names
vector of structures
nested structure
ordinal categorical arrays
descending order
ascending order
selection sort

CONTENTS

Data structures are variables that store more than one value. For it to make sense to store more than one value in a variable, the values should somehow be logically related. There are many different kinds of data structures. We have already been working with one kind, arrays (e.g., vectors and matrices). An array is a data structure in which all of the values are logically related in that they are of the same type and represent, in some sense, "the same thing". So far, that has been true for the vectors and matrices that we have used. We use vectors and matrices when we want to be able to loop through them (or, essentially, have this done for us using vectorized code).

A ***cell array*** is a kind of data structure that stores values of different types. Cell arrays can be vectors or matrices; the different values are referred to as the elements of the array. One very common use of a cell array in previous versions of the MATLAB® software was to store strings of different lengths (although as of R2016b string arrays can now be used for strings of different lengths). Cell arrays actually store pointers to the stored data.

Structures are data structures that group together values that are logically related, but are not the same thing and not necessarily the same type. The different values are stored in separate ***fields*** of the structure.

MATLAB®. https://doi.org/10.1016/B978-0-12-815479-3.00008-8

One use of structures is to set up a ***database*** of information. For example, a professor might want to store for every student in a class: the student's name, university identifier number, grades on all assignments and quizzes, and so forth. In many programming languages and database programs, the terminology is that within a database file there would be one ***record*** of information for each student; each separate piece of information (name, quiz 1 score, and so on) would be called a ***field*** of the record. In MATLAB, these records are called structures, or **structs**.

Both cell arrays and structures can be used to store values that are different types in a single variable. The main difference between them is that cell arrays are indexed and can therefore be used with loops or vectorized code. Structures, however, are not indexed; the values are referenced using the names of the fields, which is more mnemonic than indexing.

Other, more advanced, data structures are also covered in this chapter. These include ***categorical arrays*** and ***tables***. Categorical arrays are a type of array that allows one to store a finite, countable number of different possible values. A ***table*** is a data structure that stores information in a table format with rows and columns, each of which can be mnemonically labeled. An advantage of a table is that information can be extracted using either numeric indexing or by using row and variable names.

Finally, ***sorting*** the various types of data structures will be covered, both programmatically and using built-in sort functions. With a database, it is frequently useful to have it sorted on multiple fields, but this can be time-consuming. The use of ***index vectors*** is also presented as an alternative to physically sorting a database.

8.1 CELL ARRAYS

One type of data structure that MATLAB has, but is not found in many programming languages, is a ***cell array***. A cell array in MATLAB is an array, but, unlike the vectors and matrices we have used so far, elements in cell arrays are ***cells*** that can store different types of values.

8.1.1 Creating Cell Arrays

There are several ways to create cell arrays. For example, we will create a cell array in which one element will store an integer, one element will store a character, one element will store a vector, and one element will store a string. Just like with the arrays we have seen so far, this could be a 1×4 row vector, a 4×1 column vector, or a 2×2 matrix. Some of the syntax for creating vectors and matrices is the same as before in that values within rows are separated by spaces

or commas, and rows are separated by semicolons. However, for cell arrays, curly braces are used rather than square brackets. For example, the following creates a row vector cell array with four different types of values:

```
>> cellrowvec = {23, 'a', 1:2:9, "hello"}
cellrowvec =
  1×4 cell array
    {[23]}    {'a'}    {1×5 double}    {["hello"]}
```

To create a column vector cell array, the values are instead separated by semicolons:

```
>> cellcolvec = {23; 'a'; 1:2:9; "hello"}
cellcolvec =
  4×1 cell array
    {[     23] }
    {'a'       }
    {1×5 double}
    {["hello"] }
```

This method creates a 2×2 cell array matrix:

```
>> cellmat = {23, 'a'; 1:2:9, "hello"}
cellmat =
  2×2 cell array
    {[     23]}    {'a'       }
    {1×5 double}    {["hello"]}
```

The type of cell arrays is **cell**.

```
>> class(cellmat)
ans =
     'cell'
```

Another method of creating a cell array is to simply assign values to specific array elements and build it up element by element. However, as explained before, extending an array element by element is a very inefficient and time-consuming method.

It is much more efficient, if the size is known ahead of time, to preallocate the array. For cell arrays, this is done with the **cell** function. For example, to preallocate a variable *mycellmat* to be a 2×2 cell array, the **cell** function would be called as follows:

```
>> mycellmat = cell(2,2)
mycellmat =
  2×2 cell array
    {0×0 double}    {0×0 double}
    {0×0 double}    {0×0 double}
```

How to refer to each element to accomplish this will be explained next.

Note

This is a function call, so the arguments to the function are in parentheses; a matrix is created in which all of the elements are empty vectors. Then, each element can be replaced by the desired value.

8.1.2 Referring to and Displaying Cell Array Elements and Attributes

Just like with the other vectors we have seen so far, we can refer to individual elements of cell arrays. However, with cell arrays, there are two different ways to do this. The elements in cell arrays are cells. These cells can contain different types of values. With cell arrays, you can refer to the cells, or to the contents of the cells.

Using curly braces for the subscripts will reference the contents of a cell; this is called ***content indexing***. For example, this refers to the contents of the second element of the cell array *cellrowvec*; *ans* will have the type **char**:

```
> cellrowvec{2}
ans =
    'a'
```

Row and column subscripts are used to refer to the contents of an element in a matrix (again using curly braces):

```
>> cellmat{1,1}
ans =
    23
```

Values can be assigned to cell array elements. For example, after preallocating the variable *mycellmat* in the previous section, the elements can be initialized:

```
>> mycellmat{1,1} = 23
mycellmat =
  2×2 cell array
    {[      23]}    {0×0 double}
    {0×0 double}    {0×0 double}
```

Using parentheses for the subscripts references the cells; this is called ***cell indexing***. For example, this refers to the second cell in the cell array *cellrowvec*; *onec* will be a 1×1 **cell** array:

```
>> onec = cellcolvec(2)
onec =
  1×1 cell array
    {'a'}
>> class(onec)
ans =
    'cell'
```

When an element of a cell array is itself a data structure, only the type of the element is displayed when the cells are shown. For example, in the previous cell arrays, the vector is shown just as "1×5 double" (this is a high-level view of the cell array). This is what will be displayed with cell indexing; content indexing would display its contents:

```
>> cellmat(2,1)
  1×1 cell array
    {1×5 double}

>> cellmat{2,1}
ans =
     1     3     5     7     9
```

Since this results in a vector, parentheses can be used to refer to its elements. For example, the fourth element of the vector is:

```
>> cellmat{2,1}(4)
ans =
     7
```

One can also refer to subsets of cell arrays, such as in the following:

```
>> cellcolvec{2:3}
ans =
    'a'
ans =
     1     3     5     7     9
```

Note, however, that MATLAB stored *cellcolvec*{*2*} in the default variable *ans*, and then replaced that with the value of *cellcolvec*{*3*}. Using content indexing returns them as a ***comma-separated list***. However, they could be stored in two separate variables by having a vector of variables on the left side of an assignment:

```
>> [c1, c2] = cellcolvec{2:3}
c1 =
    'a'
c2 =
     1     3     5     7     9
```

Using cell indexing, the two cells would be put in a new cell array (in this case, in *ans*):

```
>> cellcolvec(2:3)
ans =
  2×1 cell array
    {'a'        }
    {1×5 double}
```

There are several methods for displaying cell arrays. The **celldisp** function displays the contents of all elements of the cell array. The function **cellplot** puts a graphical display of the cell array into a Figure Window; however, it is a high-level view and basically just displays the same information as typing the name of the variable (so, for instance, it would not show the contents of

Note

The index into the cell array is given in curly braces; parentheses are then used to refer to an element of the vector.

Note

The cell indexing using parentheses and the content indexing using curly braces is analagous to the methods for indexing into string arrays. With a string array, using parentheses refers to an individual string scalar, whereas using curly braces refers to the character vector that is stored within the string.

the vector in the previous example). In other words, it shows the cells, not their contents.

Many of the functions and operations on arrays that we have already seen also work with cell arrays. For example, here are some related to dimensioning:

```
>> length(cellrowvec)
ans =
     4

>> size(cellcolvec)
ans =
     4     1

>> cellrowvec{end}
ans =
    "hello"
```

To delete an element from a vector cell array, use cell indexing:

```
>> cellrowvec
cellrowvec =
  1×4 cell array
    {[23]}    {'a'}    {1×5 double}    {["hello"]}

>> cellrowvec(2) = []
cellrowvec =
  1×3 cell array
    {[23]}    {1×5 double}    {["hello"]}
```

For a matrix, an entire row or column can be deleted using cell indexing:

```
>> cellmat
cellmat =
  2×2 cell array
    {[      23]}    {'a'       }
    {1×5 double}    {["hello"]}

>> cellmat(1,:) = []
cellmat =
  1×2 cell array
    {1×5 double}    {["hello"]}
```

8.1.3 Storing Strings in Cell Arrays

One useful application of a cell array is to store character vectors of different lengths. As cell arrays can store different types of values, character vectors of different lengths can be stored in the elements. Prior to R2016b, this was the

preferred method for storing strings of different lengths. As of R2016b, however, string arrays are preferred.

```
>> cvnames = {'Sue', 'Cathy', 'Xavier'}
cvnames =
  1×3 cell array
    {'Sue'}    {'Cathy'}    {'Xavier'}
```

If the cell array stores only character vectors, the **strlength** function can be used to find the lengths of the character vectors:

```
>> strlength(cvnames)
ans =
     3     5     6
```

It is possible to convert from a cell array of character vectors to a string array, and vice versa. MATLAB has several functions that facilitate this. The **string** function can convert from a cell array to a string array:

```
>> sanames = string(cvnames)
sanames =
  1×3 string array
    "Sue"    "Cathy"    "Xavier"
```

The **cellstr** function will convert from a string array to a cell array of character vectors.

```
>> cellstr(sanames)
ans =
  1×3 cell array
    {'Sue'}    {'Cathy'}    {'Xavier'}
```

The function **strjoin** will concatenate, or join, all strings from a cell array into one character vector separated by one space each by default (but other delimiters can be specified).

```
>> strjoin(cvnames)
ans =
    'Sue Cathy Xavier'
>> thisday = strjoin({'January','5','2018'},'-')
thisday =
    'January-5-2018'
```

The function **strsplit** will essentially do the opposite; it splits a string into elements in a cell array with either a specified delimiter or a blank space by default.

```
>> ca = strsplit(thisday, '-')
ca =
  1×3 cell array
    {'January'}    {'5'}    {'2018'}
```

The function **iscellstr** will return **logical true** if a cell array is a cell array of all strings or **logical false** if not.

```
>> iscellstr(cvnames)
ans =
     1

>> iscellstr(cellcolvec)
ans =
     0
```

We will see several examples that utilize cell arrays containing character vectors of varying lengths in later chapters, including advanced file input functions and customizing plots.

PRACTICE 8.1

Write an expression that would display a random element from a cell array (without assuming that the number of elements in the cell array is known). Create two different cell arrays and try the expression on them to make sure that it is correct.

For more practice, write a function that will receive one cell array as an input argument and will display a random element from it.

8.2 STRUCTURES

Structures are data structures that group together values that are logically related in what are called ***fields*** of the structure. An advantage of structures is that the fields are named, which helps to make it clear what values are stored in the structure. However, structure variables are not arrays. They do not have elements that are indexed, so it is not possible to loop through the values in a structure or to use vectorized code.

8.2.1 Creating and Modifying Structure Variables

Creating structure variables can be accomplished by simply storing values in fields using assignment statements or by using the **struct** function.

In our first example, assume that the local Computer Super Mart wants to store information on the software packages that it sells. For each one, they will store the following:

- item number
- cost to the store
- price to the customer
- character code indicating the type of software

An individual structure variable for a given software package might look like this:

package

item_no	cost	price	code
123	19.99	39.95	g

The name of the structure variable is *package*; it has four fields: *item_no*, *cost*, *price*, and *code*. Note that some programmers use names that begin with an uppercase letter for structure variables (e.g., *Package*) to make them easily distinguishable.

One way to initialize a structure variable is to use the **struct** function. The names of the fields are passed as strings; each one is followed by the value for that field (so, pairs of field names and values are passed to **struct**).

```
>> package = struct('item_no',123,'cost',19.99,...
      'price',39.95,'code','g')

package =
     item_no: 123
        cost: 19.9900
       price: 39.9500
        code: 'g'

>> class(package)
ans =
    'struct'
```

MATLAB, as it is written to work with arrays, assumes the array format. Just like a single number is treated as a 1×1 double, a single structure is treated as a 1×1 struct. Later in this chapter, we will see how to work more generally with vectors of structs.

Note

In the Workspace Window, the variable *package* is listed as a 1×1 struct; the type of the variable is **struct**.

An alternative method of creating this structure, which is not as efficient, involves using the ***dot operator*** to refer to fields within the structure. The name of the structure variable is followed by a dot, or period, and then the name of the field within that structure. Assignment statements can be used to assign values to the fields.

```
>> package.item_no = 123;
>> package.cost = 19.99;
>> package.price = 39.95;
>> package.code = 'g';
```

By using the dot operator in the first assignment statement, a structure variable is created with the field *item_no*. The next three assignment statements add more fields to the structure variable. Again, extending the structure in this manner is not as efficient as using **struct**.

Adding a field to a structure later is accomplished as shown here, by using an assignment statement.

An entire structure variable can be assigned to another. This would make sense, for example, if the two structures had some values in common. Here, for example, the values from one structure are copied into another and then two fields are selectively changed, referring to them using the dot operator.

```
>> newpack = package;
>> newpack.item_no = 111;
>> newpack.price = 34.95
newpack =
    item_no: 111
       cost: 19.9900
      price: 34.9500
       code: 'g'
```

To print from a structure, the **disp** function will display either the entire structure or an individual field.

```
>> disp(package)
    item_no: 123
       cost: 19.9900
      price: 39.9500
       code: 'g'

>> disp(package.cost)
   19.9900
```

However, using **fprintf** only individual fields can be printed; the entire structure cannot be printed without referring to all fields individually.

```
>> fprintf('%d %c\n', package.item_no, package.code)
123 g
```

The function **rmfield** removes a field from a structure. It returns a new structure with the field removed, but does not modify the original structure (unless the returned structure is assigned to that variable). For example, the following would remove the *code* field from the *newpack* structure, but store the resulting structure in the default variable *ans*. The value of *newpack* remains unchanged.

```
>> rmfield(newpack, 'code')
ans =
    item_no: 111
       cost: 19.9900
      price: 34.9500
>> newpack
newpack =
    item_no: 111
       cost: 19.9000
      price: 34.9500
       code: 'g'
```

To change the value of *newpack*, the structure that results from calling **rmfield** must be assigned to *newpack*.

```
>> newpack = rmfield(newpack, 'code')
newpack =
     item_no: 111
        cost: 19.9000
       price: 34.9500
```

PRACTICE 8.2

A silicon wafer manufacturer stores, for every part in its inventory, a part number, quantity in the factory, and the cost for each.

onepart

part_no	quantity	costper
123	4	33.95

Create this structure variable using **struct**. Print the cost in the form $xx.xx.

8.2.2 Passing Structures to Functions

An entire structure can be passed to a function or individual fields can be passed. For example, here are two different versions of a function that calculates the profit on a software package. The profit is defined as the price minus the cost.

In the first version, the entire structure variable is passed to the function, so the function must use the dot operator to refer to the *price* and *cost* fields of the input argument.

calcprof.m

```
function profit = calcprof(packstruct)
% calcprofit calculates the profit for a
% software package
% Format: calcprof(structure w/ price & cost fields)

profit = packstruct.price - packstruct.cost;
end
```

```
>> calcprof(package)
ans =
    19.9600
```

In the second version, just the *price* and *cost* fields are passed to the function using the dot operator in the function call. These are passed to two scalar input arguments in the function header, so there is no reference to a structure variable in the function itself, and the dot operator is not needed in the function.

calcprof2.m

```
function profit = calcprof2(oneprice, onecost)
% Calculates the profit for a software package
% Format: calcprof2(price, cost)

profit = oneprice - onecost;
end
```

```
>> calcprof2(package.price, package.cost)
ans =
   19.9600
```

It is important, as always with functions, to make sure that the arguments in the function call correspond one-to-one with the input arguments in the function header. In the case of *calcprof*, a structure variable is passed to an input argument, which is a structure. For the second function *calcprof2*, two individual fields, which are **double** values, are passed to two **double** input arguments.

8.2.3 Related Structure Functions

There are several functions that can be used with structures in MATLAB. The function **isstruct** will return **logical** 1 for **true** if the variable argument is a structure variable or 0 if not. The **isfield** function returns **logical true** if a fieldname (as a character vector or string) is a field in the structure argument or **logical false** if not.

```
>> isstruct(package)
ans =
     1

>> isfield(package,'cost')
ans =
     1
```

The **fieldnames** function will return the names of the fields that are contained in a structure variable.

```
>> pack_fields = fieldnames(package)
pack_fields =
  4×1 cell array
    {'item_no'}
    {'cost'   }
    {'price'  }
    {'code'   }
```

As the names of the fields are of varying lengths, the **fieldnames** function returns a cell array with the names of the fields as character vectors.

Curly braces are used to refer to the elements, as *pack_fields* is a cell array. For example, we can refer to the length of one of the field names:

```
>> length(pack_fields{2})
ans =
     4
```

QUICK QUESTION!

How can we ask the user for a field in a structure and either print its value or an error if it is not actually a field?

Answer: To do this, we need to use a ***dynamic field name*** to refer to a field in the structure. A static field name is

```
struct.fieldname
```

whereas a dynamic field name uses a character vector

```
struct.('fieldname')
```

which means that the fieldname could be read in from the user. The **isfield** function can be used to determine whether or not it is a field of the structure. Then, by using a dynamic field name, we can make the code general. The following is the code for the variable *package*:

```
inputfield = input('Which field would you like to see: ','s');
if isfield(package, inputfield)
    fldtouse = package.(inputfield);
    fprintf('The value of the %s field is: ', ...
            inputfield)
    disp(fldtouse)
else
    fprintf('Error: %s is not a valid field\n', inputfield)
end
```

This code would produce this output (assuming the *package* variable was initialized as shown previously):

```
Which field would you like to see: cost
The value of the cost field is: 19.9900
```

8.2.4 Vectors of Structures

In many applications, including database applications, information would normally be stored in a ***vector of structures***, rather than in individual structure variables. For example, if the computer super mart is storing information on all of the software packages that it sells, it would likely be in a vector of structures such as the following:

packages

	item_no	cost	price	code
1	123	19.99	39.95	g
2	456	5.99	49.99	l
3	587	11.11	33.33	w

In this example, *packages* is a vector that has three elements. It is shown as a column vector. Each element is a structure consisting of four fields: *item_no*, *cost*, *price*, and *code*. It may look like a matrix, which has rows and columns, but it is, instead, a vector of structures.

This vector of structures can be created several ways. One method is to create a structure variable, as shown earlier, to store information on one software package.

This can then be expanded to be a vector of structures.

```
>> packages = struct('item_no',123,'cost',19.99,...
     'price',39.95,'code','g');
>> packages(2) = struct('item_no',456,'cost', 5.99,...
     'price',49.99,'code','l');
>> packages(3) = struct('item_no',587,'cost',11.11,...
     'price',33.33,'code','w');
```

The first assignment statement shown here creates the first structure in the structure vector, the next one creates the second structure, and so on. This actually creates a 1×3 row vector.

Alternatively, the first structure could be treated as a vector to begin with, for example

```
>> packages(1) = struct('item_no',123,'cost',19.99,...
     'price',39.95,'code','g');
>> packages(2) = struct('item_no',456,'cost', 5.99,...
     'price',49.99,'code','l');
>> packages(3) = struct('item_no',587,'cost',11.11,...
     'price',33.33,'code','w');
```

Both of these methods, however, involve extending the vector. As we have already seen, preallocating any vector in MATLAB is more efficient than extending it. There are several methods of preallocating the vector. By starting with the last element, MATLAB would create a vector with that many elements. Then, the elements from 1 through end-1 could be initialized. For example, for a vector of structures that has three elements, start with the third element.

```
>> packages(3) = struct('item_no',587,'cost',11.11,...
     'price',33.33,'code','w');
>> packages(1) = struct('item_no',123,'cost',19.99,...
     'price',39.95,'code','g');
>> packages(2) = struct('item_no',456,'cost', 5.99,...
     'price',49.99,'code','l');
```

Also, the vector of structures can be preallocated without assigning any values.

```
>> packages(3) = ...
   struct('item_no',[],'cost',[],'price',[],'code',[])
packages =
1x3 struct array with fields:
    item_no
    cost
    price
    code
```

Then, the values in the individual structures could be replaced in any order as above.

Typing the name of the variable will display only the size of the structure vector and the names of the fields:

```
>> packages
packages =
1x3 struct array with fields:
    item_no
    cost
    price
    code
```

The variable *packages* is now a vector of structures, so each element in the vector is a structure. To display one element in the vector (one structure), an index into the vector would be specified. For example, to refer to the second element:

```
>> packages(2)
ans =
    item_no: 456
       cost: 5.9900
      price: 49.9900
       code: 'l'
```

To refer to a field, it is necessary to refer to the particular structure, and then the field within it. This means using an index into the vector to refer to the structure, and then the dot operator to refer to a field. For example:

```
>> packages(1).code
ans =
    'g'
```

Thus, there are essentially three levels to this data structure. The variable *packages* is the highest level, which is a vector of structures. Each of its elements is an individual structure. The fields within these individual structures are the lowest level. The following loop displays each element in the *packages* vector.

```
>> for i = 1:length(packages)
       disp(packages(i))
   end

    item_no: 123
       cost: 19.9900
      price: 39.9500
       code: 'g'

    item_no: 456
       cost: 5.9900
      price: 49.9900
       code: 'l'

    item_no: 587
       cost: 11.1100
      price: 33.3300
       code: 'w'
```

To refer to a particular field for all structures, in most programming languages it would be necessary to loop through all elements in the vector and use the dot operator to refer to the field for each element. However, this is not the case in MATLAB.

THE PROGRAMMING CONCEPT

For example, to print all of the costs, a **for** loop could be used:

```
>> for i=1:3
       fprintf('%f\n',packages(i).cost)
   end
19.990000
5.990000
11.110000
```

THE EFFICIENT METHOD

However, **fprintf** would do this automatically in MATLAB:

```
>> fprintf('%f\n',packages.cost)
19.990000
5.990000
11.110000
```

Using the dot operator in this manner to refer to all values of a field would result in the values being stored successively in the default variable *ans* as this method results in a comma-separated list:

```
>> packages.cost
ans =
   19.9900
ans =
    5.9900
ans =
   11.1100
```

However, the values can all be stored in a vector:

```
>> pc = [packages.cost]
pc =
  19.9900    5.9900    11.1100
```

Using this method, MATLAB allows the use of functions on all of the same fields within a vector of structures. For example, to sum all three cost fields, the vector of cost fields is passed to the **sum** function:

```
>> sum([packages.cost])
ans =
   37.0900
```

For vectors of structures, the entire vector (e.g., *packages*) could be passed to a function, or just one element (e.g., *packages(1)*) which would be a structure, or a field within one of the structures (e.g., *packages(2).price*).

The following is an example of a function that receives the entire vector of structures as an input argument and prints all of it in a nice table format.

printpackages.m

```
function printpackages(packstruct)
% printpackages prints a table showing all
% values from a vector of 'packages' structures
% Format: printpackages(package structure)

fprintf('\nItem # Cost Price Code\n\n')
no_packs = length(packstruct);
for i = 1:no_packs
    fprintf('%6d %6.2f %6.2f %3c\n', ...
        packstruct(i).item_no, ...
        packstruct(i).cost, ...
        packstruct(i).price, ...
        packstruct(i).code)
end
end
```

The function loops through all of the elements of the vector, each of which is a structure, and uses the dot operator to refer to and print each field. An example of calling the function follows:

```
>> printpackages(packages)

Item #    Cost    Price    Code

   123   19.99    39.95      g
   456    5.99    49.99      l
   587   11.11    33.33      w
```

PRACTICE 8.3

A silicon wafer manufacturer stores, for every part in their inventory, a part number, how many are in the factory, and the cost for each. First, create a vector of structs called *parts* so that, when displayed, it has the following values:

```
>> parts
parts =
1x3 struct array with fields:
    partno
    quantity
    costper

>> parts(1)
ans =
       partno: 123
     quantity: 4
      costper: 33

>> parts(2)
ans =
       partno: 142
     quantity: 1
      costper: 150

>> parts(3)
ans =
       partno: 106
     quantity: 20
      costper: 7.5000
```

Next, write general code that will, for any values and any number of structures in the variable *parts*, print the part number and the total cost (quantity of the parts multiplied by the cost of each) in a column format.

For example, if the variable *parts* stores the previous values, the result would be:

```
123 132.00
142 150.00
106 150.00
```

The previous example involved a vector of structs. In the next example, a somewhat more complicated data structure will be introduced: a vector of structs in which some fields are vectors themselves. The example is a database of information that a professor might store for a course. This will be implemented as a vector of structures. The vector will store all of the course information.

Every element in the vector will be a structure, representing all information about one particular student. For every student, the professor wants to store (for now, this would be expanded later):

- name (a character vector)
- university identifier (ID) number
- quiz scores (a vector of four quiz scores)

The vector variable, called *student*, might look like the following:

student

	name	id_no	quiz			
			1	2	3	4
1	C, Joe	999	10.0	9.5	0.0	10.0
2	Hernandez, Pete	784	10.0	10.0	9.0	10.0
3	Brownnose, Violet	332	7.5	6.0	8.5	7.5

Each element in the vector is a struct with three fields (*name*, *id_no*, *quiz*). The *quiz* field is a vector of quiz grades. The *name* field is a character vector.

This data structure could be defined as follows.

```
>> student(3) = struct('name','Brownnose, Violet',...
   'id_no',332,'quiz', [7.5  6  8.5  7.5]);
>> student(1) = struct('name','C, Joe',...
   'id_no',999,'quiz', [10  9.5  0  10]);
>> student(2) = struct('name','Hernandez, Pete',...
   'id_no',784,'quiz', [10  10  9  10]);
```

Once the data structure has been initialized, in MATLAB we could refer to different parts of it. The variable *student* is the entire array; MATLAB just shows the names of the fields.

```
>> student
student =
1x3 struct array with fields:
    name
    id_no
    quiz
```

To see the actual values, one would have to refer to individual structures and/or fields.

```
>> student(1)
ans =
     name: 'C, Joe'
    id_no: 999
     quiz: [10 9.5000 0 10]

>> student(1).quiz
ans =
   10.0000  9.5000    0  10.0000

>> student(1).quiz(2)
ans =
    9.5000

>> student(3).name(1)
ans =
    'B'
```

With a more complicated data structure like this, it is important to be able to understand different parts of the variable. The following are examples of expressions that refer to different parts of this data structure:

- `student` is the entire data structure, which is a vector of structs
- `student(1)` is an element from the vector, which is an individual struct
- `student(1).quiz` is the *quiz* field from the structure, which is a vector of **double** values
- `student(1).quiz(2)` is an individual **double** quiz grade
- `student(3).name(1)` is the first letter of the third student's name (a **char**)

One example of using this data structure would be to calculate and print the quiz average for each student. The following function accomplishes this. The *student* structure, as defined before, is passed to this function. The algorithm for the function is:

- Print column headings
- Loop through the individual students; for each:
  - Sum the quiz grades
  - Calculate the average
  - Print the student's name and quiz average

With the programming method, a second (nested) loop would be required to find the running sum of the quiz grades. However, as we have seen, the **sum** function can be used to sum the vector of all quiz grades for each student. The function is defined as follows:

printAves.m

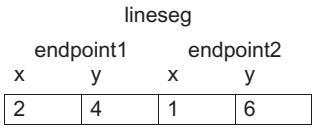
```
function printAves(student)
% This function prints the average quiz grade
% for each student in the vector of structs
% Format: printAves(student array)

fprintf('%-20s %-10s\n', 'Name', 'Average')
for i = 1:length(student)
    qsum = sum([student(i).quiz]);
    no_quizzes = length(student(i).quiz);
    ave = qsum / no_quizzes;
    fprintf('%-20s %.1f\n', student(i).name, ave);
end
```

Here is an example of calling the function:

```
>> printAves(student)
Name                 Average
C, Joe               7.4
Hernandez, Pete      9.8
Brownnose, Violet    7.4
```

8.2.5 Nested structures

A ***nested structure*** is a structure in which at least one member is itself a structure. For example, a structure for a line segment might consist of fields representing the two points at the ends of the line segment. Each of these points would be represented as a structure consisting of the x and y coordinates.



This shows a structure variable called *lineseg* that has two fields for the endpoints of the line segment, *endpoint1* and *endpoint2*. Each of these is a structure consisting of two fields for the x and y coordinates of the individual points, *x* and *y*.

One method of defining this is to nest calls to the **struct** function:

```
>> lineseg = struct('endpoint1',struct('x',2,'y',4), ...
                    'endpoint2',struct('x',1,'y',6))
```

This method is the most efficient.

Another method would be to create structure variables first for the points, and then use these for the fields in the **struct** function (instead of using another **struct** function).

```
>> pointone = struct('x', 5, 'y', 11);
>> pointtwo = struct('x', 7, 'y', 9);
```

```
>> lineseg = struct('endpoint1', pointone,...
                    'endpoint2', pointtwo);
```

A third method, the least efficient, would be to build the nested structure one field at a time. As this is a nested structure with one structure inside of another, the dot operator must be used twice here to get to the actual *x* and *y* coordinates.

```
>> lineseg.endpoint1.x = 2;
>> lineseg.endpoint1.y = 4;
>> lineseg.endpoint2.x = 1;
>> lineseg.endpoint2.y = 6;
```

Once the nested structure has been created, we can refer to different parts of the variable *lineseg*. Just typing the name of the variable shows only that it is a structure consisting of two fields, *endpoint1* and *endpoint2*, each of which is a structure.

```
>> lineseg
lineseg =
    endpoint1: [1x1 struct]
    endpoint2: [1x1 struct]
```

Typing the name of one of the nested structures will display the field names and values within that structure:

```
>> lineseg.endpoint1
ans =
    x: 2
    y: 4
```

Using the dot operator twice will refer to an individual coordinate, such as in the following example:

```
>> lineseg.endpoint1.x
ans =
     2
```

QUICK QUESTION!

How could we write a function *strpoint* that returns a string "(x,y)" containing the x and y coordinates? For example, it might be called separately to create strings for the two endpoints and then printed as shown here:

```
>> fprintf('The line segment consists of %s and %s\n',...
   strpoint(lineseg.endpoint1), ...
   strpoint(lineseg.endpoint2))
The line segment consists of (2, 4) and (1, 6)
```

QUICK QUESTION!—CONT'D

Answer: As an *endpoint* structure is passed to an input argument in the function, the dot operator is used within the function to refer to the x and y coordinates. The **sprintf** function is used to create the string that is returned.

`strpoint.m`

```
function ptstr = strpoint(ptstruct)
% strpoint receives a struct containing x and y
% coordinates and returns a string '(x,y)'
% Format: strpoint(structure with x and y fields)

ptstr = sprintf("(%d, %d) ", ptstruct.x, ptstruct.y);
end
```

8.2.6 Vectors of Nested Structures

Combining vectors and nested structures, it is possible to have a vector of structures in which some fields are structures themselves. Here is an example in which a company manufactures cylinders from different materials for industrial use. Information on them is stored in a data structure in a program. The variable *cyls* is a vector of structures, each of which has fields *code*, *dimensions*, and *weight*. The *dimensions* field is a structure itself consisting of fields *rad* and *height* for the radius and height of each cylinder.

cyls

	code	dimensions		weight
		rad	height	
1	x	3	6	7
2	a	4	2	5
3	c	3	6	9

The following is an example of initializing the data structure by preallocating:

```
>> cyls(3) = struct('code', 'c', 'dimensions',...
    struct('rad', 3, 'height', 6), 'weight', 9);
>> cyls(1) = struct('code', 'x', 'dimensions',...
    struct('rad', 3, 'height', 6), 'weight', 7);
>> cyls(2) = struct('code', 'a', 'dimensions',...
    struct('rad', 4, 'height', 2), 'weight', 5);
```

There are several layers in this variable. For example:

- `cyls` is the entire data structure, which is a vector of structs
- `cyls(1)` is an individual element from the vector, which is a struct
- `cyls(2).code` is the *code* field from the struct *cyls(2)*; it is a **char**

- *cyls(3).dimensions* is the *dimensions* field from the struct *cyls(3)*; it is a struct itself
- *cyls(1).dimensions.rad* is the *rad* field from the struct *cyls(1).dimensions*; it is a **double** number

For these cylinders, one desired calculation may be the volume of each cylinder, which is defined as $\pi * r^2 * h$, where r is the radius and h is the height. The following function *printcylvols* prints the volume of each cylinder, along with its code for identification purposes. It calls a local function to calculate each volume.

printcylvols.m

```
function printcylvols(cyls)
% printcylvols prints the volumes of each cylinder
% in a specialized structure
% Format: printcylvols(cylinder structure)

% It calls a local function to calculate each volume

for i = 1:length(cyls)
    vol = cylvol(cyls(i).dimensions);
    fprintf('Cylinder %c has a volume of %.1f in^3\n', ...
       cyls(i).code, vol);
end
end

function cvol = cylvol(dims)
% cylvol calculates the volume of a cylinder
% Format: cylvol(dimensions struct w/ fields 'rad', 'height')

cvol = pi * dims.rad ^2 * dims.height;
end
```

The following is an example of calling this function.

```
>> printcylvols(cyls)
Cylinder x has a volume of 169.6 in^3
Cylinder a has a volume of 100.5 in^3
Cylinder c has a volume of 169.6 in^3
```

Note

The entire data structure, *cyls*, is passed to the function. The function loops through every element, each of which is a structure. It prints the *code* field for each, which is given by *cyls(i).code*. To calculate the volume of each cylinder, only the radius and height are needed, so rather than passing the entire structure to the local function *cylvol* (which would be *cyls(i)*), only the *dimensions* field is passed (*cyls(i).dimensions*). The function then receives the *dimensions* structure as an input argument and uses the dot operator to refer to the *rad* and *height* fields within it.

PRACTICE 8.4

Modify the function *cylvol* to calculate and return the surface area of the cylinder in addition to the volume (2 pi r^2 + 2 pi r h).

8.3 ADVANCED DATA STRUCTURES

MATLAB has several types of data structures in addition to the arrays, cell arrays, and structures that we have already seen. These can be found in the Documentation under Data Types.

8.3.1 Categorical Arrays

Categorical arrays are a type of array that allows one to store a finite, countable number of different possible values. Categorical arrays are fairly new in MATLAB, introduced in R2013b. Categorical arrays are defined using the **categorical** function.

For example, a group is polled on their favorite ice cream flavors; the results are stored in a categorical array:

```
>> icecreamfaves = categorical({'Vanilla', 'Chocolate', ...
'Chocolate', 'Rum Raisin', 'Vanilla', 'Strawberry', ...
'Chocolate', 'Rocky Road', 'Chocolate', 'Rocky Road', ...
'Vanilla', 'Chocolate', 'Strawberry', 'Chocolate'});
```

Another way to create this would be to store the strings in a cell array, and then convert using the **categorical** function:

```
>> cellicecreamfaves = {'Vanilla', 'Chocolate', ...
'Chocolate', 'Rum Raisin', 'Vanilla', 'Strawberry', ... 'Chocolate',
'Rocky Road', 'Chocolate', 'Rocky Road', ...
'Vanilla', 'Chocolate', 'Strawberry', 'Chocolate'}
>> icecreamfaves = categorical(cellicecreamfaves);
```

There are several functions that can be used with categorical arrays. The function **categories** will return the list of possible categories as a cell column vector, sorted in alphabetical order.

```
>> cats = categories(icecreamfaves)
cats =
  4×1 cell array
    {'Chocolate' }
    {'Rum Raisin' }
    {'Strawberry'}
    {'Vanilla'   }
```

The functions **countcats** and **summary** will show the number of occurrences of each of the categories.

```
>> countcats(icecreamfaves)
ans =
     6     2     1     2     3
```

```
>> summary(icecreamfaves)
     Chocolate  Rocky Road  Rum Raisin  Strawberry  Vanilla
             6           2           1           2        3
```

In the case of the favorite ice cream flavors, there is no natural order for them, so they are listed in alphabetical order. It is also possible to have ***ordinal categorical arrays***, however, in which an order is given to the categories.

For example, a person has a wearable fitness tracker that tracks the days on which a personal goal for the number of steps taken is reached; these are stored in a file. To simulate this, a variable *stepgoalsmet* stores these data for a few weeks. Another cell array stores the possible days of the week.

```
>> stepgoalsmet = {'Tue', 'Thu', 'Sat', 'Sun', 'Tue', ...
'Sun', 'Thu', 'Sat', 'Wed', 'Sat', 'Sun'};
>> daynames = {'Mon','Tue','Wed','Thu','Fri','Sat','Sun'};
```

Then, an ordinal categorical array, *ordgoalsmet*, is created. This allows days to be compared using relational operators.

```
>> ordgoalsmet = categorical(stepgoalsmet,daynames,'Ordinal',true);
>> summary(ordgoalsmet)
     Mon     Tue     Wed     Thu     Fri     Sat     Sun
       0       2       1       2       0       3       3
>> ordgoalsmet(1) < ordgoalsmet(3)
ans =
     1
>> ordgoalsmet(4) < ordgoalsmet(3)
ans =
     0
```

8.3.2 Tables

A ***table*** is a data structure that stores information in a table format with rows and columns, each of which can be mnemonically labeled. The table is created using variables that have the same length and might be stored in columns, for example, in a spreadsheet. For example, the following uses the **table** function to store some simple information for a doctor's patients. There are just three patients, so there are three names in one variable that will be used to refer to the rows, and the heights and weights of the patients are stored in column vectors (each of which has three values). The column vectors do not need to store the same types of values, but they must be of the same length.

```
>> names = {'Harry','Sally','Jose'};
>> weights = [185; 133; 210]; % Note column vectors
>> heights = [74; 65.4; 72.2];
>> patients = table(weights, heights, 'RowNames', names)
patients =
  3×2 table
```

```
              weights    heights
              _______    _______
    Harry     185          74
    Sally     133        65.4
    Jose      210        72.2
```

This created a 3×2 table, with two variables named *weights* and *heights*.

There are many ways to index into tables, to either create a new table that is a subset of the original, or to extract information from the table into other types of data structures. Using parentheses, we can index into this table to get a subset of the table, which would also be a table. The indexing can be done using integers (as with arrays we have seen so far) or by using row or variable names.

```
>> patients(1:2, 1)
ans =
              weights
              _______
    Harry     185
    Sally     133
>> patients({'Harry' 'Jose'}, :)
ans =
              weights    heights
              _______    _______
    Harry     185        74
    Jose      210        72.2
```

Using curly braces to index, the data can be extracted; in the following example, into a **double** matrix or column vector.

```
>> mat = patients{{'Harry' 'Jose'}, :}
mat =
  185.0000   74.0000
  210.0000   72.2000
>> wtcol = patients{:, 'weights'}
wtcol =
   185
   133
   210
>> mat = patients{:, 1}
mat =
   185
   133
   210
```

Notice that either numerical indices can be used, or the more mnemonic row or variable names. Tables combine the mnemonic names that structures have with the ability to index that is found in arrays.

The **summary** function can be used for tables; it shows the variables and some statistical data for each.

```
>> summary(patients)
Variables:
    weights: 3x1 double
        Values:
            min       133
            median    185
            max       210
    heights: 3x1 double
        Values:
            min       65.4
            median    72.2
            max         74
```

8.4 SORTING

Sorting is the process of putting a list in order—either ***descending*** (highest to lowest) or ***ascending*** (lowest to highest) order. For example, here is a list of *n* integers, visualized as a column vector.

1	85
2	70
3	100
4	95
5	80
6	91

What is desired is to sort this in ascending order in place—by rearranging this vector, not creating another. The following is one basic algorithm.

- Look through the vector to find the smallest number and then put it in the first element in the vector. How? By exchanging it with the number currently in the first element.
- Then, scan the rest of the vector (from the second element down) looking for the next smallest (or, the smallest in the rest of the vector). When found, put it in the first element of the rest of the vector (again, by exchanging).
- Continue doing this for the rest of the vector. Once the next-to-last number has been placed in the correct location in the vector, the last number, by default, has been as well.

What is important in each pass through the vector is *where* the smallest value is, so the elements to be exchanged are known (not what the actual smallest number is).

This table shows the progression. The left column shows the original vector. The second column (from the left) shows that the smallest number, the 70, is now in the first element in the vector. It was put there by exchanging with what had been in the first element, 85. This continues element-by-element, until the vector has been sorted.

85	70	70	70	70	70
70	85	80	80	80	80
100	100	100	85	85	85
95	95	95	95	91	91
80	80	85	100	100	95
91	91	91	91	95	100

This is called the ***selection sort***; it is one of many different sorting algorithms.

THE PROGRAMMING CONCEPT

The following function implements the selection sort to sort a vector:

mysort.m

```
function outv = mysort(vec)
% mysort sorts a vector using the selection sort
% Format: mysort(vector)

% Loop through the elements in the vector to end-1
for i = 1:length(vec)-1
    indlow = i; % stores the index of the smallest
    % Find where the smallest number is
    %    in the rest of the vector
    for j=i+1:length(vec)
        if vec(j) < vec(indlow)
            indlow = j;
        end
    end
    % Exchange elements
    temp = vec(i);
    vec(i) = vec(indlow);
    vec(indlow) = temp;
end
outv = vec;
end
```

```
>> vec = [85 70 100 95 80 91];
>> vec = mysort(vec)
vec =
    70     80     85     91     95     100
```

THE EFFICIENT METHOD

MATLAB has a built-in function, **sort**, which will sort a vector in ascending order:

```
>> vec = [85 70 100 95 80 91];
>> vec = sort(vec)
vec =
    70    80    85    91    95     100
```

Descending order can also be specified. For example,

```
>> sort(vec,'descend')
ans =
  100    95    91    85    80    70
```

Sorting a row vector results in another row vector. Sorting a column vector results in another column vector. Note that if we did not have the 'descend' option, **flip** could be used after sorting.

For matrices, the **sort** function will by default sort each column. To sort by rows, the dimension 2 is specified. For example,

```
>> mat
mat =
    4     6     2
    8     3     7
    9     7     1
>> sort(mat) % sorts by column
ans =
    4     3     1
    8     6     2
    9     7     7
>> sort(mat,2) % sorts by row
ans =
    2     4     6
    3     7     8
    1     7     9
```

8.4.1 Sorting Vectors of Structures

When working with a vector of structures, it is common to sort based on a particular field of the structures. For example, recall the vector of structures used to store information on different software packages which was created in Section 8.2.4.

packages

	item_no	cost	price	code
1	123	19.99	39.95	g
2	456	5.99	49.99	l
3	587	11.11	33.33	w

Here is a function that sorts this vector of structures in ascending order based on the *price* field.

mystructsort.m

```
function outv = mystructsort(structarr)
% mystructsort sorts a vector of structs on the price field
% Format: mystructsort(structure vector)

for i = 1:length(structarr)-1
    indlow = i;
    for j=i+1:length(structarr)
        if structarr(j).price < structarr(indlow).price
           indlow = j;
        end
    end
    % Exchange elements
    temp = structarr(i);
    structarr(i) = structarr(indlow);
    structarr(indlow) = temp;
end
outv = structarr;
end
```

Recall that we created a function *printpackages* that prints the information in a nice table format. Calling the *mystructsort* function and also the function to print will demonstrate this:

```
>> printpackages(packages)
Item #   Cost   Price   Code

   123   19.99   39.95   g
   456    5.99   49.99   l
   587   11.11   33.33   w

>> packByPrice = mystructsort(packages);
>> printpackages(packByPrice)

Item #   Cost   Price   Code

   587   11.11   33.33   w
   123   19.99   39.95   g
   456    5.99   49.99   l
```

Note

Only the *price* field is compared in the sort algorithm, but the entire structure is exchanged. Consequently, each element in the vector, which is a structure of information about a particular software package, remains intact.

This function only sorts the structures based on the *price* field. A more general function is shown in the following, which receives a string that is the name of the field. The function checks first to make sure that the string that is passed is a valid field name for the structure. If it is, it sorts based on that field, and if not, it returns an empty vector. The function uses dynamic field names to refer to the field in a structure.

generalPackSort.m

```
function outv = generalPackSort(inputarg, fname)
% generalPackSort sorts a vector of structs
% based on the field name passed as an input argument

if isfield(inputarg,fname)
   for i = 1:length(inputarg)-1
       indlow = i;
       for j=i+1:length(inputarg)
           if [inputarg(j).(fname)] < ...
              [inputarg(indlow).(fname)]
               indlow = j;

           end
       end
       % Exchange elements
       temp = inputarg(i);
       inputarg(i) = inputarg(indlow);
       inputarg(indlow) = temp;
   end
   outv = inputarg;
else
   outv = [];
end
end
```

The following are examples of calling the function:

```
>> packByPrice = generalPackSort(packages,'price');
>> printpackages(packByPrice)

Item #   Cost   Price   Code

   587   11.11   33.33   w
   123   19.99   39.95   g
   456    5.99   49.99   l

>> packByCost = generalPackSort(packages,'cost');
>> printpackages(packByCost)
```

```
Item #   Cost   Price   Code

   456    5.99   49.99   l
   587   11.11   33.33   w
   123   19.99   39.95   g

>> packByProfit = generalPackSort(packages,'profit')
packByProfit =
      []
```

QUICK QUESTION!

Is this *generalPackSort* function truly general? Would it work for any vector of structures, not just one configured like *packages*?

Answer: It is fairly general. It will work for any vector of structures. However, the comparison will only work for numerical or character fields. Thus, as long as the field is a number or character, this function will work for any vector of structures. If the field is a vector itself (including a character vector), it will not work.

8.4.2 Sorting Strings

For a string array, the sort function will sort the strings alphabetically.

```
>> wordarr = ["Hello" "Howdy" "Hi" "Goodbye" "Ciao"];
>> sort(wordarr)
ans =
  1×5 string array
     "Ciao"    "Goodbye"    "Hello"    "Hi"    "Howdy"
```

For a character matrix, the **sort** function sorts column by column, using the ASCII equivalents of the characters. It can be seen from the results that the space character comes before the letters of the alphabet in the character encoding:

```
>> words = char('Hello', 'Howdy', 'Hi', 'Goodbye', 'Ciao')
words =
  5×7 char array
     'Hello  '
     'Howdy  '
     'Hi     '
     'Goodbye'
     'Ciao   '
>> sort(words)
ans =
  5×7 char array
```

```
    'Ce      '
    'Giad    '
    'Hildb   '
    'Hoolo   '
    'Howoyye'
```

This is one reason that a character matrix is not a good idea! However, there is a way to sort these words, which are rows in the matrix. MATLAB has a function **sortrows** that will do this. The way it works is that it examines the row column by column starting from the left. If it can determine which letter comes first, it picks up the entire row and puts it in the first row. In this example, the first two rows are placed based on the first character, 'C' and 'G'. For the other three strings, they all begin with 'H' so the next column is examined. In this case, the rows are placed based on the second character, 'e', 'i', 'o'.

```
>> sortrows(words)
ans =
  5×7 char array
    'Ciao   '
    'Goodbye'
    'Hello  '
    'Hi     '
    'Howdy  '
```

The **sortrows** function sorts each row as a block, or group, and it will also work on numbers. In this example, the rows beginning with 3 and 4 are placed first. Then, for the rows beginning with 5, the values in the second column (6 and 7) determine the order.

```
>> mat = [5 7 2; 4 6 7; 3 4 1; 5 6 2]
mat =
     5     7     2
     4     6     7
     3     4     1
     5     6     2
>> sortrows(mat)
ans =
     3     4     1
     4     6     7
     5     6     2
     5     7     2
```

The **sortrows** function can also be used to sort rows in a table.

```
>> patients
patients =
```

```
  3×2 table
              weights    heights
              _______    _______
    Harry       185         74
    Sally       133       65.4
    Jose        210       72.2
>> sortrows(patients,'heights')
ans =
  3×2 table
              weights    heights
              _______    _______
    Sally       133         65.4
    Jose        210         72.2
    Harry       185           74
```

The function **issortedrows** can be used to determine whether the rows in a matrix or table are sorted, as they would be with **sortrows**.

In order to sort a cell array of character vectors, the **sort** function can be used. If the cell array is a row vector, a sorted row vector is returned and, if the cell array is a column vector, a sorted column vector is returned. For example, note the transpose operator below which makes this a column vector.

```
>> engcellnames = {'Chemical','Mechanical',...
     'Biomedical','Electrical', 'Industrial'};
>> sort(engcellnames')
ans =
  5×1 cell array
    {'Biomedical'}
    {'Chemical'  }
    {'Electrical'}
    {'Industrial'}
    {'Mechanical'}
```

Categorical arrays can also be sorted using the **sort** function. A non-ordinal categorical array, such as *icecreamfaves*, will be sorted in alphabetical order.

```
>> sort(icecreamfaves)
ans =
Chocolate     Chocolate     Chocolate     Chocolate
Chocolate     Chocolate     Rocky Road    Rocky Road
Rum Raisin    Strawberry    Strawberry    Vanilla
Vanilla       Vanilla
```

An ordinal categorical array, however, will be sorted using the order specified. For example, the ordinal categorical array *ordgoalsmet* was created using *daynames*:

```
>> ordgoalsmet
ordgoalsmet =
     Tue      Thu      Sat      Sun      Tue      Sun
     Thu      Sat      Wed      Sat      Sun

>> daynames = ...
       {'Mon','Tue','Wed','Thu','Fri','Sat','Sun'};
```

Thus, the sorting is done in the order given by *daynames*.

```
>> sort(ordgoalsmet)
ans =
    Tue      Tue      Wed      Thu      Thu      Sat
    Sat      Sat      Sun      Sun      Sun
```

8.5 INDEX VECTORS

Using ***index vectors*** is an alternative to sorting a vector. With indexing, the vector is left in its original order. An index vector is used to "point" to the values in the original vector in the desired order.

For example, here is a vector of exam grades:

grades

1	2	3	4	5	6
85	70	100	95	80	91

In ascending order, the lowest grade is in element 2, the next lowest grade is in element 5, and so on. The index vector *grade_index* gives the order for the vector *grades*.

grade_index

1	2	3	4	5	6
2	5	1	6	4	3

The elements in the index vector are then used as the indices for the original vector. To get the *grades* vector in ascending order, the indices used would be *grades(2)*, *grades(5)*, and so on. Using the index vector to accomplish this, `grades(grade_index(1))` would be the lowest grade, 70, and `grades(grade_index(2))` would be the second-lowest grade. In general, `grades(grade_index(i))` would be the *i*th lowest grade.

To create these in MATLAB:

```
>> grades = [85 70 100 95 80 91];
>> grade_index = [2 5 1 6 4 3];
>> grades(grade_index)
ans =
    70    80    85    91    95   100
```

Note that this is a particular type of index vector in which all of the indices of the original vector appear, in the desired order.

In general, instead of creating the index vector manually as shown here, the procedure to initialize the index vector is to use a sort function. The following is the algorithm:

- initialize the values in the index vector to be the indices 1,2, 3, … to the length of the original vector
- use any sort algorithm, but compare the elements in the original vector using the index vector to index into it (e.g., using *grades(grade_index(i))* as previously shown)
- when the sort algorithm calls for exchanging values, exchange the elements in the index vector, not in the original vector

Here is a function that implements this algorithm:

createind.m

```
function indvec = createind(vec)
% createind returns an index vector for the
% input vector in ascending order
% Format: createind(inputVector)

% Initialize the index vector
len = length(vec);
indvec = 1:len;

for i = 1:len-1
    indlow = i;
    for j=i+1:len
        % Compare values in the original vector
        if vec(indvec(j)) < vec(indvec(indlow))
            indlow = j;
        end
    end
   % Exchange elements in the index vector
   temp = indvec(i);
   indvec(i) = indvec(indlow);
   indvec(indlow) = temp;
end
end
```

For example, for the grades vector just given:

```
>> clear grade_index
>> grade_index = createind(grades)
grade_index =
     2     5     1     6     4     3
```

```
>> grades(grade_index)
ans =
    70    80    85    91    95   100
```

8.5.1 Indexing Into Vectors of Structures

Often, when the data structure is a vector of structures, it is necessary to iterate through the vector in order by different fields. For example, for the *packages* vector defined previously, it may be necessary to iterate in order by the *cost* or by the *price* fields.

Rather than sorting the entire vector of structures based on these fields, it may be more efficient to index into the vector based on these fields; so, for example, to have an index vector based on *cost* and another based on *price*.

packages

	item_no	cost	price	code
1	123	19.99	39.95	g
2	456	5.99	49.99	l
3	587	11.11	33.33	w

	cost_ind
1	2
2	3
3	1

	price_ind
1	3
2	1
3	2

These index vectors would be created as before, comparing the fields, but exchanging the entire structures. Once the index vectors have been created, they can be used to iterate through the *packages* vector in the desired order. For example, the function to print the information from *packages* has been modified so that, in addition to the vector of structures, the index vector is also passed and the function iterates using that index vector.

printpackind.m

```
function printpackind(packstruct, indvec)
% printpackind prints a table showing all
% values from a vector of packages structures
% using an index vector for the order
% Format: printpackind(vector of packages, index vector)

fprintf('Item # Cost Price Code\n')
no_packs = length(packstruct);
for i = 1:no_packs
    fprintf('%6d %6.2f %6.2f %3c\n', ...
        packstruct(indvec(i)).item_no, ...
        packstruct(indvec(i)).cost, ...
        packstruct(indvec(i)).price, ...
        packstruct(indvec(i)).code)
end
end
```

```
>> printpackind(packages,cost_ind)
Item #   Cost   Price    Code
   456    5.99   49.99     l
   587   11.11   33.33     w
   123   19.99   39.95     g

>> printpackind(packages,price_ind)
Item #   Cost   Price   Code
   587   11.11   33.33   w
   123   19.99   39.95   g
   456    5.99   49.99   l
```

PRACTICE 8.5

Modify the function *createind* to create the *cost_ind* index vector.

■ Explore Other Interesting Features

The built-in functions **cell2struct**, which converts a cell array into a vector of structs, and

struct2cell, which converts a struct to a cell array.

Find the functions that convert from cell arrays to number arrays and vice versa.

Explore the **orderfields** function.

MATLAB has an entire category of data types and built-in functions that operate on dates and times. Find this under Language Fundamentals, then Data Types.

Explore the "is" functions for categorical arrays, such as **iscategorical**, **iscategory**, and **isordinal**.

Explore the table functions **array2table** and **struct2table**.

Explore **timetable**, which is a table in which every row is time-stamped.

Explore the functions **deal** (which assigns values to variables) and **orderfields**, which puts structure fields in alphabetical order.

Investigate the **randperm** function. ■

SUMMARY

COMMON PITFALLS

- Confusing the use of parentheses (cell indexing) versus curly braces (content indexing) for a cell array
- Forgetting to index into a vector using parentheses or referring to a field of a structure using the dot operator
- When sorting a vector of structures on a field, forgetting that although only the field in question is compared in the sort algorithm, entire structures must be interchanged.

PROGRAMMING STYLE GUIDELINES

- Use arrays when values are of the same type and represent in some sense the same thing.
- Use cell arrays or structures when the values are logically related, but not of the same type nor the same thing.
- Use cell arrays, rather than structures, when it is desired to loop through the values or to vectorize the code.
- Use structures rather than cell arrays when it is desired to use names for the different values rather than indices.
- Use **sortrows** to sort strings stored in a matrix alphabetically; for cell arrays and string arrays, **sort** can be used.
- When it is necessary to iterate through a vector of structures in order based on several different fields, it may be more efficient to create index vectors based on these fields rather than sorting the vector of structures multiple times.

MATLAB Functions and Commands

cell	strsplit	isfield	summary
celldisp	iscellstr	fieldnames	table
cellplot	struct	categorical	sort
cellstr	rmfield	categories	sortrows
strjoin	isstruct	countcats	issortedrows

MATLAB Operators

cell arrays { }
dot operator for structs .
parentheses for dynamic field names ( )

Exercises

1. Create the following cell array:

```
>> ca = {'abc', 11, 3:2:9, zeros(2)}
```

   Use the **reshape** function to make it a *2* × *2* matrix. Then, write an expression that would refer to just the last column of this cell array.
2. Create a *2* × *2* cell array using the **cell** function and then put values in the individual elements. Then, insert a row in the middle so that the cell array is now *3* × *2*.
3. Create a row vector cell array to store the character vector 'xyz', the number 33.3, the vector 2:6, and the **logical** expression 'a' < 'c'. Use the transpose operator to make this a column vector, and use **reshape** to make it a 2 × 2 matrix. Use **celldisp** to display all elements.
4. Create a cell array that stores phrases, such as:

```
exclaimcell = {'Bravo', 'Fantastic job'};
```

   Pick a random phrase to print.
5. Create three cell array variables that store people's names, verbs, and nouns. For example,

```
names = {'Harry', 'Xavier', 'Sue'};
verbs = {'loves', 'eats'};
nouns = {'baseballs', 'rocks', 'sushi'};
```

   Write a script that will initialize these cell arrays, and then print sentences using one random element from each cell array (e.g., 'Xavier eats sushi').
6. Write a script that will prompt the user for character vectors and read them in, store them in a cell array (in a loop), and then print them out.
7. When would you loop through the elements of a cell array?
8. Vectorize the following code! Rewrite this as just one statement that accomplishes the same end result. You may assume that *mycvs* is a cell array containing only character vectors, and that all have been initialized.

```
newstr = mycvs{1};
for i = 2:length(mycvs)
 newstr = [newstr, ' ', mycvs{i}];
end
newstr % Note just for display
```

9. Write a function *buildstr* that will receive a character and a positive integer n. It will create and return a cell array with character vectors of increasing lengths, from 1 to the integer *n*. It will build the character vectors with successive characters in the ASCII encoding.

```
>> buildstr('a',4)
ans =
   'a'   'ab'   'abc'   'abcd'
```

10. Create a cell array that contains character vectors, e.g.,

```
pets = {'cat', 'dog', 'snake'};
```

Show the difference in the following methods of indexing:

```
pets(1:2)
pets{1:2}
[p1 p2] = pets{1:2}
```

11. Write a function *caprint* that will receive one input argument, which is a cell array (you may assume that the input argument is in fact a cell array). If the cell array contains only character vectors, the function will print each in the following format; otherwise, it prints an error message.

```
>> fishies = {'tuna','shark','salmon'};
>> caprint(fishies)
String 1 is: tuna
String 2 is: shark
String 3 is: salmon

>> myca = {2:5, 'hi', 'hello'};
>> caprint(myca)
Sorry, not all strings!
```

12. Create a cell array variable that would store for a student his or her name, university id number, and GPA. Print this information.
13. Create a structure variable that would store for a student his or her name, university id number, and GPA. Print this information.
14. Here is an inefficient way of creating a structure variable to store the three parts of a person's phone number:

```
>> myphone.area = '803';
>> myphone.loc = '878';
>> myphone.num = '9876';
```

Rewrite this more efficiently using the **struct** function:

15. What would be an advantage of using cell arrays over structures?
16. What would be an advantage of using structures over cell arrays?
17. A complex number is a number of the form a + ib, where a is called the real part, b is called the imaginary part, and i = . Write a script that prompts the user separately to enter values for the real and imaginary parts and stores them in a structure variable. It then prints the complex number in the form a + ib. The script should just print the value of a, then the character vector '+i', and then the value of b. For example, if the script is named *compnumstruct*, running it would result in:

```
>> compnumstruct
Enter the real part: 2.1
```

```
Enter the imaginary part: 3.3
The complex number is 2.1 + i3.3
```

18. Create a data structure to store information about the elements in the periodic table of elements. For every element, store the name, atomic number, chemical symbol, class, atomic weight, and a seven-element vector for the number of electrons in each shell. Create a structure variable to store the information, for example, for lithium:

```
Lithium 3 Li alkali_metal 6.94 2 1 0 0 0 0 0
```

19. Write a function *separatethem* that will receive one input argument which is a structure containing fields named 'length' and 'width', and will return the two values separately. Here is an example of calling the function:

```
>> myrectangle = struct('length',33,'width',2);
>> [l w] = separatethem(myrectangle)
l =
  33
w =
   2
```

20. Write a function *prtnames* that will receive a struct as an input and will print the names of the fields from the struct in the format shown below. You may assume that the input argument that is passed to the function is in fact a struct.

```
>> st = struct('month',3,'day',24);
>> prtnames(st)
Field 1 is: 'month'
Field 2 is: 'day'
```

21. A script stores information on potential subjects for an experiment in a vector of structures called *subjects*. The following shows an example of what the contents might be:

```
>> subjects(1)
ans =
    name: 'Joey'
  sub_id: 111
  height: 6.7000
  weight: 222.2000
```

For this particular experiment, the only subjects who are eligible are those whose height or weight is lower than the average height or weight of all subjects. The script will print the names of those who are eligible. Create a vector with sample data in a script, and then write the code to accomplish this. Don't assume that the length of the vector is known; the code should be general.

22. Quiz data for a class is stored in a file. Each line in the file has the student ID number (which is an integer) followed by the quiz scores for that student. For example, if there are four students and three quizzes for each, the file might look like this:

```
44      7       7.5     8
33      5.5     6       6.5
37      8       8       8
24      6       7       8
```

First create the data file, and then store the data in a script in a vector of structures. Each element in the vector will be a structure that has 2 members: the integer student ID number, and a vector of quiz scores. The structure will look like this:

students

	id_no	quiz 1	2	3
1	44	7	7.5	8
2	33	5.5	6	6.5
3	37	8	8	8
4	24	6	7	8

To accomplish this, first use the **load** function to read all information from the file into a matrix. Then, using nested loops, copy the data into a vector of structures as specified. Then, the script will calculate and print the quiz average for each student.

23. Create a nested struct to store a person's name, address, and phone numbers. The struct should have 3 fields for the name, address, and phone. The address fields and phone fields will be structs.
24. Design a nested structure to store information on constellations for a rocket design company. Each structure should store the constellation's name and information on the stars in the constellation. The structure for the star information should include the star's name, core temperature, distance from the sun, and whether it is a binary star or not. Create variables and sample data for your data structure.
25. Write a script that creates a vector of line segments (where each is a nested structure as shown in this chapter). Initialize the vector using any method. Print a table showing the values, such as shown in the following:

```
Line   From      To
====   =======   =======
  1    ( 3, 5)   ( 4, 7)
  2    ( 5, 6)   ( 2, 10)
       etc.
```

26. Given a vector of structures defined by the following statements:

```
kit(2) = struct('sub', ...
      struct('id',123,'wt',4.4,'code','a'),...
      'name', 'xyz', 'lens', [4 7])
kit(1) = struct('sub', ...
      struct('id',33,'wt',11.11,'code','q'), ...
      'name', 'rst', 'lens', 5:6)
```

Which of the following expressions are valid? If the expression is valid, give its value. If it is not valid, explain why.

```
>> kit(1).sub
>> kit(2).lens(1)
>> kit(1).code
>> kit(2).sub.id == kit(1).sub.id
>> strfind(kit(1).name, 's')
```

27. A team of engineers is designing a bridge to span the Podunk River. As part of the design process, the local flooding data must be analyzed. The following information on each storm that has been recorded in the last 40 years is stored in a file: a code for the location of the source of the data, the amount of rainfall (in inches), and the duration of the storm (in hours), in that order. For example, the file might look like this:

```
321  2.4  1.5
111  3.3  12.1
   etc.
```

Create a data file. Write the first part of the program: design a data structure to store the storm data from the file, and also the intensity of each storm. The intensity is the rainfall amount divided by the duration. Write a function to read the data from the file (use **load**), copy from the matrix into a vector of structs, and then calculate the intensities. Write another function to print all of the information in a neatly organized table. Add a function to the program to calculate the average intensity of the storms. Add a function to the program to print all of the information given on the most intense storm. Use a subfunction for this function which will return the index of the most intense storm.

28. Create an ordinal categorical array to store the four seasons.
29. Create a categorical array to store the favorite breakfast beverage of your close friends. Show the results from using the **countcats** and **summary** functions on your categorical array.
30. Create a **table** to store information on students; for each, their name, id number, and major.
31. Sort your table from Exercise 30 on the id numbers, in descending order.

32. Write a function *mydsort* that sorts a vector in descending order (using a loop, not the built-in sort function).
33. Write a function *matsort* to sort all of the values in a matrix (decide whether the sorted values are stored by row or by column). It will receive one matrix argument and return a sorted matrix. Do this without loops, using the built-in functions **sort** and **reshape**. For example:

```
>> mat
mat =

     4     5     2
     1     3     6
     7     8     4
     9     1     5

>> matsort(mat)
ans =

     1     4     6
     1     4     7
     2     5     8
     3     5     9
```

34. DNA is a double-stranded helical polymer that contains basic genetic information in the form of patterns of nucleotide bases. The patterns of the base molecules A, T, C, and G encode the genetic information. Construct a cell array to store some DNA sequences as character vectors; such as

    TACGGCAT
    ACCGTAC

    and then sort these alphabetically. Next, construct a matrix to store some DNA sequences of the same length and then sort them alphabetically.

35. Trace this; what will it print?

parts

	code	quantity	weight
1	'x'	11	4.5
2	'z'	33	3.6
3	'a'	25	4.1
4	'y'	31	2.2

	ci
1	3
2	1
3	4
4	2

	qi
1	1
2	3
3	4
4	2

	wi
1	4
2	2
3	3
4	1

```
for i = 1:length(parts)
   fprintf('Part %c weight is %.1f\n',...
        parts(qi(i)).code, parts(qi(i)).weight)
end
```

36. When would you use sorting vs indexing?
37. Write a function that will receive a vector and will return two index vectors: one for ascending order and one for descending order. Check the function by writing a script that will call the function and then use the index vectors to print the original vector in ascending and descending order.
38. Create a cell array of all character vectors. Sort them alphabetically. Investigate the use of some string functions on the cell array, e.g., **lower**, **count**, and **contains**.
39. The **wordcloud** function (introduced in 2017b) can be used to create a word cloud from a categorical array or from a table. Create a categorical array and get a word cloud from it, e.g.,

```
>> icecreamfaves = categorical({'Vanilla', 'Chocolate', ...
'Chocolate', 'Rum Raisin', 'Vanilla', 'Strawberry', ...
'Chocolate', 'Rocky Road', 'Chocolate', 'Rocky Road', ...
'Vanilla', 'Chocolate', 'Strawberry', 'Chocolate'});
>> wordcloud(icecreamfaves)
```