CHAPTER 7

Cell Arrays and Structures

Chapter Objectives

This chapter discusses the nature, implementation, and behavior of collections that may contain data items of any class, size, or shape. We will deal with two different heterogeneous storage mechanisms:

- Those accessed by index (cell arrays)
- Those accessed by field name (structures)

In addition, we will consider collecting structures into arrays of structures.

Introduction

This chapter covers data collections that are more general and flexible than the arrays we have considered so far. Heterogeneous collections may contain objects of any type, rather than just numbers. Consequently, none of the collective operations defined for numerical arrays can be applied to cell arrays or structures. To perform most operations on their contents, the items must be extracted one at a time and replaced if necessary. We will consider three different mechanisms for building heterogeneous collections: you access components of a cell array with a numerical index; you access components of a structure with a symbolic field name; and you access components of a structure array by way of a numerical index to reach a specific structure, and then a symbolic field name.

7.1 Concept: Collecting Dissimilar Objects

Heterogeneous collections permit objects of different data types to be grouped in a collection. They allow data abstraction to apply to a much broader range of content. However, the fact that the contents of these collections may be of any data type severely restricts the operations that can be performed on the collections as a whole. Whereas a significant number of arithmetic and logical operations can be performed on whole number arrays, algorithms that process heterogeneous collections almost always deal with the data contents one item at a time.

7.2 Cell Arrays

Cell arrays, as the name suggests, have the general form of arrays and can be indexed numerically as arrays. However, each element of a cell array should be considered as a container in which one data object of any class can be stored.[1] They can be treated as arrays of containers for the purpose of concatenation and slicing. However, if you wish to access or modify the contents of the containers, the cells must be accessed individually.

7.2.1 Creating Cell Arrays

Cell arrays may be constructed in the following ways:

- By assigning values individually to a variable indexed with braces:

```
>> A{1} = 42
A =
    [42]
```

- By assigning containers individually to a variable indexed with brackets:

```
>> B[1] = {[4 6]};
B =
    [1x2 double]
```

- By concatenating cell contents using braces {. . .}:

```
C = {3, [1,2,3], 'abcde'}
C =
    [3] [1x3 double] 'abcde'
```

- By concatenating cell containers:

```
>> D = [A B C {'xyz'}]
D =
    [42] [1x2 double] [3] [1x3 double] 'abcde' 'xyz'
```

[1] Java programmers might recognize a cell array as an array of Objects.

Based on these examples, we observe the following:

- A cell array can contain any legal MATLAB object
- Just as with number arrays, cell arrays can be created "on the fly" by assigning values to an indexed variable

When the values from a cell array are displayed, their appearance is different from that of the contents of a number array. Individual numbers are shown in brackets, for example, `[3]`; larger numerical arrays display their size, for example, `[1x3 double]`; and character strings are displayed with the enclosing quotes, for example, `'abcde'`.

7.2.2 Accessing Cell Arrays

Since cell arrays can be considered as conventional arrays of containers, the containers can be accessed and manipulated normally. For example, continuing the previous examples, we have the following:

```
>> E = D(2) % parentheses - a container
E =
    [4 6]
```

However, braces are used to access the contents of the containers as follows:

```
>> D{2} % braces - the contents
ans =
    4 6
```

If the right-hand side of an assignment statement results in multiple cell arrays, the assignment must be to the same number of variables. The built-in function `deal(...)` is used to make these allocations. Exercise 7.1 shows its use.

Notice the following observations:

- When we extract the contents of multiple cells using `A{1:2}`, this results in multiple assignments being made. These multiple assignments must go to separate variables. This is the fundamental mechanism behind returning multiple results from a function.
- These multiple assignments cannot be made to a single variable; sufficient storage must be provided either as a collection of variables or explicitly as a vector.
- Cell arrays can be "sliced" with normal vector indexing assignments as long as the sizes match on the left and right sides of the assignment. Any unassigned array elements are filled with an empty vector.
- The assignment `B{[1 3]} = A{[1 2]}` that produced an error needs some thought. Since `A{[1 2]}` produces two separate assignments, MATLAB will not assign the answers, even to the right number of places in another cell array. The `deal(...)` function is provided to capture these multiple results in different variables. Notice the difference between `A{:}` and `A` as a parameter to `deal(...)`. When

Exercise 7.1 Cell arrays

```
>> A = { 3, [1,2,3] 'abcde'}
 A =
     [3] [1x3 double] 'abcde'
>> A{1:2}
 ans =
     3
 ans =
      1    2    3
>> [x y] = A{1:2}
x =
     3
y =
     1    2    3
>> B = A{1:2}
 ??? Illegal right-hand side in assignment.
           Too many elements.
>> B([1 3]) = A([1 2])
 B =
   [3]     []    [1x3 double]
>> B{[1 3]} = A{[1 2]}
 ??? Illegal right-hand side in assignment.
           Too many elements.
>> [a, b, c] = deal(A{:})
 a =
     3
 b =
      1    2    3
 c =
 abcde
>> [a, b] = deal(A)
 a =
     [3]    [1x3 double]    'abcde'
 b =
     [3]    [1x3 double]    'abcde'
>> B = A(1:2)
 B =
     [3]    [1x3 double]
>> for i = 1:2
          s(i) = sum(A{i})
       end
s =
     3
s =
     3    6
>> F{2} = 42
F =
    []     [42]
>> F{3} = {42}
F =
    []     [42]    {1x1 cell}
```

`deal(...)` is provided with a parameter other than a collection of cells, it copies that parameter to each variable.

- Assignments work normally if cell arrays are treated as vectors and the extraction of items can be indexed—`s` is a vector of the sums of the elements in `A`.
- Finally, notice that when accessing cell arrays, it is normal to have braces on one side or the other of an assignment; it is rarely appropriate to have braces on both sides of an assignment. The result here is that a cell array is loaded into the third container in the cell array.

7.2.3 Using Cell Arrays

There are a number of uses for cell arrays, some of which will be evident in upcoming chapters. For now, the following examples will suffice:

- Containing lists of possible values for `switch/case` statements, as we saw in Chapter 4
- Substituting for parameter lists in function calls

For example, suppose you have a function `largest(a, b, c)` that consumes three variables and produces the largest of the three values provided. It can be used in the following styles, as shown in Listing 7.1.

In Listing 7.1:

Lines 1–3: Set the values of `A`, `B`, and `C`.

Line 4: A conventional function call that results in a value of 6 for `N`.

Lines 5–6: The same function call implemented as a cell array, returning the same answer.

7.2.4 Processing Cell Arrays

The general template for processing cell arrays is shown in Template 7.1.

Checking the class of the element can be achieved in one of two ways:

- The function `class(item)` returns a string specifying the item type that can be used in a `switch` statement

Listing 7.1 Using cell arrays of parameters

```
1. A = 4;
2. B = 6;
3. C = 5;
4. N = largest(A, B, C)
5. params = { 4, 6, 5 };
6. N = largest(params{1:3})
```

Template 7.1 General template for processing cell arrays

```
<initialize result>
for <index specification>
      <extract an element>
      <check the element accordingly>
      <process the element accordingly>
end
<finalize result>
```

Listing 7.2 Cell array processing example

```
1. function ans = totalNums(ca)
   % count the numbers in a cell array
2.     ans = 0 ;
3.     for in = 1 :length(ca)
4.         item = ca{i} ;     % extract the item
5.         if isnumeric(item) % check if a vector
6.             ans = ans + prod(size(item));
7.         end
8.     end
```

- Individual test functions can be used in an `if... elseif` construct; examples of the individual test functions are `isa(item, 'class')`, `iscell(...)`, `ischar(...)`, `islogical(...)`, `isnumeric(...)`, and `isstruct(...)`.

For example, suppose you are provided with a cell array and have been asked for a function that finds the total length of all the vectors it contains. The function might look like that shown in Listing 7.2.

In Listing 7.2:

Line 1: Typical function header accepting a cell array as input.

Line 2: Initializes the result.

Line 3: Traverses the whole cell array.

Line 4: Extracts each item in turn.

Line 5: Determines whether this item is of type `double`. If so, it proceeds to line 6.

Line 6: Accumulates the number of items in this array. Recall that the `size(...)` function returns a vector of the sizes of each dimension. The total number of numbers is therefore the product of these values.

7.3 Structures

Where cell arrays implemented the concept of homogeneous collections as indexed collections, structures allow items in the collection to be indexed by field name. Most modern languages implement the concept of a structure in a similar style. The data contained in a structure are referenced by field

name, for example, `item1`. The rules for making a field name are the same as those for a variable. Fields of a structure, like the elements of a cell array, are heterogeneous—they can contain any MATLAB object. First, we will see how to construct and manipulate one structure, and then how to aggregate individual structures into an array of structures.

7.3.1 Constructing and Accessing One Structure

To set the value of items in a structure `A`, the syntax is as follows:

```
>> A.item1 = 'abcde'
 A =
    item1: 'abcde'
>> A.item2 = 42
A =
    item1: 'abcde'
    item2: 42
```

Notice that MATLAB displays the elements of an emerging structure by name. Fields in a structure are accessed in the same way—by using the dotted notation.

```
>> A.item2 = A.item2 ./ 2
 A =
    item1: 'abcde'
    item2: 21
```

You can determine the names of the fields in a structure using the built-in function `fieldnames(...)`. It returns a cell array containing the field names as strings.

```
>> names = fieldnames(A)
 names =
    'item1'
    'item2'
```

Fields can also be accessed "indirectly" by setting a variable to the name of the field, and then by using parentheses to indicate that the variable contents should be used as the field name:

```
>> fn = names{1};
 >> A.(fn) = [A.(fn) 'fg']
A =
     item1: 'abcdefg'
     item2: 21
```

Common Pitfalls 7.1

Be careful. `rmfield(...)` returns a new structure with the requested field removed. It does not remove that field from your original structure. If you want the field removed from the original, you must assign the result from `rmfield(...)` to replace the original structure:

```
>> A = rmfield(A, 'item1')
A =
     item2: 21
```

You can also remove a field from a structure using the built-in function `rmfield(...)`. Exercise 7.2 gives you an opportunity to understand how to build structures. Here we build a typical structure that could be used as one entry in a telephone book. Since phone numbers usually contain punctuation, we could store

Exercise 7.2 Building structures

Suppose that you want to use structures to maintain your address book. In the Command window, enter the following commands:

```
>> entry.first = 'Fred'
entry =
    first: 'Fred'
>> entry.last = 'Jones';
>> entry.phone = '(123) 555-1212'
entry =
    first: 'Fred”
     last: 'Jones'
    phone: '(123) 555-1212'
>> entry.phone
ans =
 (123) 555-1212
>> date.day = 31;
>> date.month = 'February';
>> date.year = 1965
date =
      day: 31
    month:'February'
     year: 1965
>> entry.birth = date
entry =
    first: 'Fred'
     last: 'Jones'
    phone: '(123) 555-1212'
    birth: [1x1 struct]
>> entry.birth
ans =
      day: 31
    month: 'February'
     year: '1965'
>> entry.birth.year
ans =
  1965
```

them as strings. Notice that since a structure may contain any object, it is quite legal to make a structure containing a date and insert that structure in the date field of the entry. The structure display function, however, does not display the contents of the structures.

7.3.2 Constructor Functions

This section discusses functions that assign their parameters to the fields of a structure and then return that structure. You do this, as opposed to “manually” entering data into structures, for the following reasons:

- Manual entry can result in strange behavior due to typographical errors or having fields in the wrong order

- The resulting code is generally more compact and easier to understand
- When constructing collections of structures, it enforces consistency across the collections

There are two approaches to the use of constructor functions: using built-in capabilities and writing your own constructor. There is a built-in function, `struct(...)`, that consumes pairs of entries (each consisting of a field name as a string and a cell array of field contents) and produces a structure. If all the cell arrays have more than one entry, this actually creates a structure array, as discussed in Section 7.4.1.

The following command would construct the address book entry created in the previous section. Note the use of ellipses `(...)` to indicate to the MATLAB machinery that the logic is continued onto the next line.

```
>> struct('first','Fred', ...
'last','Jones', ...
'phone','(123) 555-1212', ...
'birth', struct( 'day', 31,
                 'month', 'February',
                 'year', 1965 ))
ans =
    first: 'Fred'
     last: 'Jones'
    phone: '(123) 555-1212'
    birth: [1x1 struct]
```

This is useful in general to create structures, but the need to repeat the field names makes this general-purpose approach a little annoying. We can create a special-purpose function that "knows" the necessary field names to create multiple structures in an organized way.

Listing 7.3 shows the code for a function that consumes parameters that describe a CD and assembles a structure containing those attributes by name.

In Exercise 7.3, you can try your hand at using this function to construct a CD structure and then verify the structure contents.

Listing 7.3 Constructor for a CD structure

```
1. function ans = makeCD(gn, ar, ti, yr, st, pr)
    % integrate CD data into a structure
2.     ans.genre = gn ;
3.     ans.artist = ar ;
4.     ans.title = ti;
5.     ans.year = yr;
6.     ans.stars = st;
7.     ans.price = pr;
```

Exercise 7.3 A CD structure

Create one entry of CD information:

```
>> CD = makeCD('Blues', 'Charles, Ray', ...
'Genius Loves Company', 2004, 4.5, 15.35 )
CD =
     genre: 'Blues'
    artist: 'Charles, Ray'
     title: 'Genius Loves Company'
      year: 2004
     stars: 4.5000
     price: 15.3500
>> flds = fieldnames(CD)
flds =
    'genre'
    'artist'
    'title'
    'year'
    'stars'
    'price'
>> field = flds{2}
field =
artist
>> CD.(field)
ans =
     Charles, Ray
```

7.4 Structure Arrays

To be useful, collections like address books or CD collections require multiple structure entries with the same fields. This is accomplished by forming an array of data items, each of which contains the same fields of information.

MATLAB implements the concept of structure arrays with the properties described in the following paragraphs.

7.4.1 Constructing Structure Arrays

Structure arrays can be created either by creating values for individual fields, as shown in Exercise 7.4; by using MATLAB's `struct(...)` function to build the whole structure array, as shown in Listing 7.4; or by using a custom function to create each individual structure, as shown in Listing 7.5. This latter listing illustrates these concepts by implementing a collection of CDs as a structure array using the function `makeCD(...)` from Listing 7.3.

Exercise 7.4 Building a structure array "by hand"

```
>> entry(1).first = 'Fred';
>> entry(1).last = 'Jones';
>> entry(1).age = 37;
>> entry(1).phone = ' (123) 555-1212';
>> entry(2).first = 'Sally';
>> entry(2).last = 'Smith';
>> entry(2).age = 29;
>> entry(2).phone = '(000) 555-1212'
entry =
1x2 structure array with fields:
     first
     last
     age
     phone
```

Listing 7.4 Building a structure array using `struct(...)`

```
 1. genres = {'Blues', 'Classical', 'Country' };
 2. artists = {'Clapton, Eric', 'Bocelli, Andrea', 'Twain, Shania' };
 3. years = { 2004, 2004, 2004 };
 4. stars = { 2, 4.6, 3.9 };
 5. prices = { 18.95, 14.89, 13.49 };
 6. cds = struct( 'genre', genres, ...
 7.                    'artist', artists, ...
 8.                    'year', years, ...
 9.                    'stars', stars, ...
10.                   'price', prices)
```

In Listing 7.4:

Lines 1–5: Build cell arrays containing field values for five CDs.

Line 6: Uses the built-in `struct(...)` function to create the CD collection. The function consumes a variable number of pairs of parameters. The first parameter of the pair is a string containing the name of a field to be created. The second parameter is the content of that field expressed as either a cell array or any other data type. If the field content is a cell array, the structure to be created becomes a structure array whose length is the length of that cell array. Each field of the structure array receives the corresponding value from the cell array. If the field content is anything other than a cell array, the content of each structure array field becomes a copy of that item.

Listing 7.5 Building a structure array using a custom constructor

```
     % extracts from http://www.cduniverse.com/   12/30/04
 1. cds(1) = makeCD('Blues', 'Clapton, Eric', ...
 2.  'Sessions For Robert J', 2004, 2, 18.95 );
 3. cds(2) = makeCD('Classical', ...
 4.  'Bocelli, Andrea', 'Andrea', 2004, 4.6, 14.89 );
 5. cds(3) = makeCD( 'Country', 'Twain, Shania', ...
 6.  'Greatest Hits', 2004, 3.9, 13.49 );
 7. cds(4) = makeCD('Latin', 'Trevi, Gloria', ...
 8.  'Como Nace El Universo', 2004, 5, 12.15 );
 9. cds(5) = makeCD('Rock/Pop', 'Ludacris', ...
10.  'The Red Light District', 2004, 4, 13.49 );
11. cds(6) = makeCD('R & B', '2Pac', ...
12.  'Loyal To The Game', 2004, 3.9, 13.49 );
13. cds(7) = makeCD('Rap', 'Eminem', ...
14.  'Encore', 2004, 3.5, 15.75 );
15. cds(8) = makeCD( 'Heavy Metal', 'Rammstein', ...
16.  'Reise, Reise', 2004, 4.2, 12.65 )
```

In Listing 7.5:

Lines 1–2: Call the `makeCD(...)` function defined in Listing 7.3 to generate the description of the first CD.

Lines 3–16: Repeat the process for seven more CDs, each of which is added to the collection.

7.4.2 Accessing Structure Elements

Like normal arrays or cell arrays, items can be stored and retrieved by their index in the array. As structures are added to the array, MATLAB forces all elements in the structure array to implement the same field names in the same order. Elements can be accessed either manually (not recommended) or by creating new structures with a constructor and adding them (recommended).

If you elect to manipulate them manually, you merely identify the array element by indexing and use the `.field` operator. For example, for the CD collection `cds`, we could change the price of one of them as follows:

```
>> cds(3).price = 11.95
 cds =
1x31 struct array with fields:
    genre,
    artist,
    title,
    year,
    stars,
    price
```

This is a little hazardous when making manual additions to a structure array. A typographical error while entering a field name results in all

the structures having that bad field name. For example, consider this error:

```
>> cds(3).prce = 11.95
 cds =
1x31 struct array with fields:
    genre,
    artist,
    title,
    year,
    stars,
    price,
    prce
```

You have accidentally added a new field to the whole collection. You can check this by looking at one entry:

```
>> cds(1)
 ans =
     genre: 'Blues'
    artist: 'Sessions For Robert J'
     title: 'Clapton, Eric'
      year: 2004
     stars: 2
     price: 18.95
      prce: []
```

If this happens, you can use the `fieldnames(...)` function to determine the situation and then the `rmfield(...)` function to remove the offending entry.

```
>> fieldnames(cds)
 ans =

    'genre'
    'artist'
    'title'
    'year'
    'stars'
    'price'
    'prce'

>> cds = rmfield(cds,'prce')
cds =
1x32 struct array with fields:
    genre,
    artist,
    title,
    year,
    stars,
    price
```

It is best to construct a complete structure and then insert it into the structure array. For example:

```
>> newCD = makeCD( 'Oldies', 'Greatest Hits', ...
 'Ricky Nelson', 2005, 5, 15.79 );
>> cds(8) = newCD
```

```
cds =
1x8 struct array with fields:
    genre,
    artist,
    title,
    year,
    stars,
    price
```

If you insert that new CD beyond the end of the array, as one might expect, MATLAB fills out the array with empty structures:

```
>> cds(50) = newCD
 cds =
1x50 struct array with fields:
    genre,
    artist,
    title,
    year,
    stars,
    price
>> cds(49)
ans =
     genre: []
    artist: []
     title: []
      year: []
     stars: []
     price: []
```

7.4.3 Manipulating Structures

Structures and structure arrays can be manipulated in the following ways:

I. Single values can be changed using the "." (dot) notation directly with a field name:

```
>> cds(5).price = 19.95;
```

II. or indirectly using the "." (dot) notation with a variable containing the field name:

```
>> fld = 'price';
>> cds(5).(fld) = 19.95;
```

or by using built-in functions:

III. `nms = fieldnames(str)` returns a cell array containing the names of the fields in a structure or structure array.

```
>> flds = fieldnames(cds)
```

Common Pitfalls 7.2

A few very understandable but sneaky errors occur when adding structures that have been created "manually" rather than by means of a standardized constructor function. If the new structure has fields not in the original structure, or extra fields, you see a slightly obscure error: `"Subscripted assignment between dissimilar structures."`

Perhaps more puzzling, if you are using an older version of MATLAB, this same error occurs if all the fields are present, but are in the wrong order.

IV. `it = isfield(str, <fldname>)` determines whether the given name is a field in this structure or structure array.

```
>> if isfield(cds, 'price') ...
```

V. `str = setfield(str, <fldname>, <value>)` returns a new structure array with the specified field set to the specified value.

```
>> cds(1) = setfield(cds(1), 'price', 19.95);
```

VI. `val = getfield(str, <fldname>)` returns the value of the specified field.

```
>> disp(getfield(cds(1), 'price') );
```

VII. `str = rmfield(str, <fldname>)` returns a new structure array with the specified field removed.

```
>> noprice = rmfield(cds, 'price');
```

VIII. Values across the whole array can be retrieved using the "." notation by accumulating them into arrays either into cell arrays:

```
>> titles = {cds.title};
>> [alpha order] = sort(titles);
```

IX. or, if the values are all numeric, into a vector:

```
>> prices = [cds.price];
>> total = sum(prices);
```

Notice that after extracting the price values into a cell array or vector, all the normal operations—in this case, `sort(...)` and `sum(...)`—can be utilized.

Exercise 7.5 provides some practice in manipulating structure arrays using the above CD collection as an example.

Exercise 7.5 The CD collection

Retrieve and run the script named `buildCDs.m` from the Companion Web site. Then, in the Interactions window, enter the following commands to create your collection of CD information:

```
>> cds(5)
ans =
     genre: 'Rock/Pop'
    artist: 'Ludacris'
     title: 'The Red Light District'
      year: 2004
     stars: 4
     price: 13.49
>> flds = fieldnames(collection)
flds =
    'genre'
    'artist'
    'title'
    'year'
    'stars'
    'price'
```

continued on next page

```
cds(5).strs = 0.5;
>> cds(5)
ans =
     genre: 'Rock/Pop'
    artist: 'Ludacris'
     title: 'The Red Light District'
      year: 2004
     stars: 4
     price: 13.4900
      strs: 0.5
>> cds(1)
ans =
     genre: 'Blues'
    artist: 'Clapton, Eric'
     title: 'Sessions For Robert J'
      year: 2004
     stars: 2
     price: 18.9500
      strs: []
>> cds = rmfield(cds, 'strs');
>> cds(1)
ans =
     genre: 'Blues'
    artist: 'Clapton, Eric'
     title: 'Sessions For Robert J'
      year: 2004
     stars: 2
     price: 18.9500
>> sum([cds.price])
ans =
  409.1100
```

7.5 Engineering Example—Assembling a Physical Structure

Many large buildings today have steel frames as their basic structure. Engineers perform the analysis and design work for each steel component and deliver these designs to the steel company. The steel company manufactures all the components, and prepares them for delivery to the building site. At this point, each component is identified only by a unique identifier string stamped and/or chalked onto that component. For even a modest-sized building, this transportation may require a significant number of truckloads of components. The question we address here is how to decide the sequence in which the components are delivered to the building site so that components are available when needed, but not piled up waiting to be used.

Consider the relatively simple structure shown in Figure 7.1. The components have individual labels, and we can obtain from the architect the identities of the components that are connected together. The

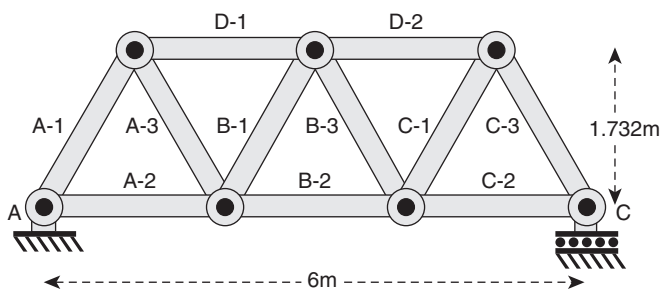

Figure 7.1 *Simple structure assembly*

construction needs to start from the fixed point A. We need to analyze this information and compute the order in which the components would be used to assemble the structure.

The data will be organized as a structure array with one entry for each component. One of the fields in that structure will be a cell array of the names of the components to which this component is connected.

The code in Listing 7.6 shows the solution to this problem.

Listing 7.6 Connectivity of a structure

```
data(1) = beam('A-1', 0.866, 0.5, ...
           {'A','A-2','A-3','D-1'} );
data(2) = beam('A-2', 0, 1, ...
           {'A', 'A-3', 'B-1', 'B-2'} );
data(3) = beam('A-3', 0.866, 1.5, ...
           {'A-1', 'A-2', 'B-1', 'D-1'} );
data(4) = beam('B-1', 0.866, 2.5, ...
           {'A-2', 'A-3', 'B-2', 'B-3', 'D-1', 'D-2'} );
data(5) = beam('B-2', 0, 3, ...
           {'A-2', 'A-3', 'B-1', 'B-3', 'C-1', 'C-2'} );
data(6) = beam('B-3', 0.866, 3.5, ...
           {'B-1', 'B-2', 'C-1', 'C-2', 'D-1', 'D-2'} );
data(7) = beam('C-1', 0.866, 4.5, ...
           {'B-2', 'B-3', 'C-2', 'C-3', 'D-2'} );
data(8) = beam('C-2', 0, 5, ...
           {'B-2', 'B-3', 'C-1', 'C-3', 'C'} );
data(9) = beam('C-3', 0.866, 5.5, ...
           {'C-1', 'C-2', 'D-2', 'C'} );
data(10) = beam('D-1', 1.732, 2, ...
           {'A-1', 'A-3', 'B-1', 'B-3', 'D-2'} );
data(11) = beam( 'D-2', 1.732, 4, ...
           {'B-1', 'B-3', 'C-1', 'C-3', 'D-1'} )
conn = 'A';
clist = {conn};
while true
```

continued on next page

```
26.     index = 0;
        % find all the beams connected to conn
27.     for in = 1:length(data)
28.         str = data(in);
29.         if touches(str, conn)
30.             index = index + 1;
31.             found(index) = str;
32.         end
33.     end
        % eliminate those already connected
34.     for jn = index:-1:1
35.         if ison(found(jn).name, clist)
36.             found(jn) = [];
37.         else
38.            clist = [clist {found(jn).name}];
39.         end
40.     end
41.     if length(found) > 0
42.         conn = nextconn( found, clist );
43.     else
44.         break;
45.     end
46. end
47. disp('the order of assembly is:')
48. disp(clist)
```

In Listing 7.6:

Lines 1–22: Construct the structure array using the `beam(...)` constructor function below.

Line 23: The current connection point, `conn`—originally, the point A.

Line 24: Initializes the connection list, a cell array of names.

Line 25: An infinite loop to be exited with break statements.

Lines 26–33: Traverse the components to make a structure array, `found`, containing all the components connected to the current connection point, `conn`.

Lines 34–40: Go through the `found` array, removing any component already on the connected list and appending the names of those not removed to the connected list.

Lines 41–45: We will exit the `while` loop when there are no new components found; until then, choose the next component to connect.

The support functions for this script are assembled for convenience into Listing 7.7. They should be in separate files with the appropriate file names to be accessible by MATLAB.

Listing 7.7 Support functions

```
1. function ans = beam( nm, xp, yp, conn )
   % construct a beam structure with fields:
   % name - beam name
   % xp, yp - coordinates of its centroid
   % conn - cell array - names of adjacent beams
   % useage: ans = beam( nm, xp, yp, conn )
2.     ans.name = nm;
3.     ans.pos = [xp, yp];
4.     ans.connect = conn;
5. end
6. function res = touches(beam, conn)
   % does the beam touch this connecting point?
   % usage: res = touches(beam, conn)
7.     res = false;
8.     for in = 1:length(beam.connect)
9.         item = beam.connect{in};
10.        if strcmp(item,conn)
11.            res = true; break;
12.        end
13.    end
14. end
15. function res = ison( nm, cl )
    % is this beam on the connection list,
    % a cell array of beam names
    % usage: res = ison( beam, cl )
16.    res = false;
17.    for in = 1:length(cl)
18.        item = cl{in};
19.        if strcmp(item, nm)
20.            res = true; break;
21.        end
22.    end
23. end
24. function nm = nextconn( fnd, cl )
    % find a connection name among
    % those found not already connected
    % usage: nm = nextconn( fnd, cl )
25.    for in = 1:length(fnd)
26.        item = fnd(in);
27.        cn = item.connect;
28.        for jn = 1:length(cn)
29.            nm = cn{jn};
30.            if ~ison(nm, cl)
31.                break;
32.            end
33.        end
34.    end
35. end
```

In Listing 7.7:

Lines 1–5: Constructor for one structure defining one component.

Lines 6–14: A function to determine whether a beam touches this connecting point.

Lines 15–23: A similar function to determine whether a particular string is on the connection list, a cell array of strings.

Lines 24–35: Function to find the next connection to use based on the latest components found—the "outer edges" of the emerging structure—and its not being already on the connected list.

Here is the resulting output:

```
data =

1x11 struct array with fields:
    name,
    pos,
    connect

the order of assembly is:
'A' 'A-2' 'A-1' 'D-1' 'A-3' 'B-2' 'B-1' 'D-2' 'B-3' 'C-2' 'C-1' 'C-3'
```

Chapter Summary

This chapter covered the nature, implementation, and behavior of two heterogeneous collections:

- Cell arrays are vectors of containers; their elements can be manipulated either as vectors of containers, or individually by inserting or extracting the contents of the container using braces in place of parentheses
- The elements of a structure are accessed by name rather than by indexing, using the dot operator, '.', to specify the field name to be used
- Structures can be collected into structure arrays whose elements are structures all with the same field names. These elements can then be indexed and manipulated in the same manner as the cells in a cell array

Special Characters, Reserved Words, and Functions

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`{ ... }`	Defines a cell array	7.2
`.<field>`	Used to access fields of a structure	7.3.1
`.(<variable>)`	Allows a variable to be used as a structure field	7.3.1
`class(<object>)`	Determines the data type of an object	7.2.4
`deal(...)`	Distributes cell array results among variables	7.2.2
`getfield (<str>, <fld>)`	Extracts the value of the field `<fld>` from a structure	7.4.3

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`isa(<object>, <class>)`	Determines whether the `<object>` is of the given data type, `<class>`	7.2.4
`iscell(<object>)`	Determines whether `<object>` is of type `cell`	7.2.4
`ischar(<object>)`	Determines whether `<object>` is of type `char`	7.2.4
`isfield(<str>, <fld>)`	`true` if the string `<fld>` is a field in the structure `<str>`	7.4.3
`islogical (<object>)`	Determines whether `<object>` is of type `logical`	7.2.4
`isnumeric (<object>)`	Determines whether `<object>` is of type `double`	7.2.4
`isstruct (<object>)`	Determines whether `<object>` is of type `struct`	7.2.4
`str = setfield(<str>, <fld>, <value>)`	Constructs a new structure that is a copy of `<str>` in which the value of the field `<fld>` has been changed to `<value>`	7.4.3
`[values order] = sort(<object>)`	Sorts either vectors (increasing numerical order) or cell arrays of strings (alphabetically) returning the sorted data and the index order for the sort	7.4.3
`struct(...)`	Constructs a structure from `<fieldname>` `<value>` pairs of parameters	7.3.2

Self Test

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

True or False

1. Of all the collective operations defined for numerical arrays, only logical operations can be applied to a whole cell array.
2. A cell array or a structure can contain any legal MATLAB object.
3. You gain access to the contents of a cell by using braces, `{...}`.
4. Since the contents of a structure are heterogeneous, we can store other structures in any structure.
5. The statement `rmfield(str, 'price')` removes the field `'price'` and its value from the structure `str`.
6. The statement `getfield(str, <fldname>)` returns the value of the specified field.

7. You cannot extract and process all of the values of a field in a structure array.

Fill in the Blanks

1. To perform any operations on the contents of a heterogeneous collection, the items must be _____ and if necessary, _____.
2. Cell arrays can be treated for the purpose of concatenation and slicing as _____of _____ .
3. The assignment `B{3} = {42}` results in the third entry in the cell array B being a(n) _____.
4. If a variable called `field` contains the name of a field in a structure `str`, the expression _____ will set the value of that field to 42.
5. MATLAB has a built-in function _____ that consumes pairs of entries, each consisting of a(n) _____and a(n) _____, and produces a structure array.

Programming Projects

1. Write a function named `cellParse` that takes in a cell array with each element being either a string (character array), or a vector (containing numbers), or a boolean value (logical array of length 1).

   Your function should return the following:

   - `nStr:` the number of strings
   - `nVec:` the number of vectors
   - `nBool:` the number of boolean values
   - `cString:` a cell array of all the strings in alphabetical order
   - `vecLength:` the average length of all the vectors
   - `allTrue:` true if all the boolean values are `true` and `false` otherwise

   For example,

   ```
   [a b c d e f] =
   cellParse( { [1 2 3], true, 'hi there!',
                        42, false, 'abc'} )
   ```

   should return `a = 2, b = 2, c = 2, d = {'abc','hi there!},` `e = 2, and f = false.`

2. It turns out that since you have become an expert on rating clothing (Chapter 4, Problem 5), Acme Clothing Company has hired you to

rate its clothes. Clothes are now represented as structures instead of vectors with the fields (all of which are numbers between 0 and 5):

Condition, Color, Price, Matches, and Comfort

Acme has a much simpler way of rating its clothes than you used before:

Rating = 5 * Condition + 3 * Color + 2 * Price + Matches
+ 9 * Comfort

You have a script called `makeClothes.m` that will create a structure array called `acmeClothes` that contains clothes structures. You are to write a script called `rateClothes` that will add a `Rating` field and a `Quality` field to each of the structures in the `acmeClothes` array. The Rating field in each structure should contain the rating of that particular article of clothing. The `Quality` field is a string that is `'premium'` if the `Rating` is over 80, `'good'` over 60, `'poor'` over 20, and `'liquidated'` for anything else.

Note:

a. You MUST use iteration to solve this problem.

b. To make things easy, just place the line `makeClothes` at the top of your script, so you're guaranteed to have the correct `acmeClothes` array to work with.

c. The fields are case sensitive, so make sure that you capitalize them.

3. You have been hired by a used-car dealership to modify the price of cars that are up for sale. You will get the information about a car, and then change its price tag depending on a number of factors.

Write a function called `usedCar` that takes in a structure with the following fields:

`Make:` A string that represents the make of the car (e.g., `'Toyota Corolla'`)

`Year:` A number that corresponds to the year of the car (e.g., `1997`)

`Cost :` A number that holds the marked price of the car (e.g., `7,000`)

`Miles :` The number of miles clocked (e.g., `85,000`)

`Accidents:` The number of accidents the car has been in (e.g., `1`)

Your function should return a structure with all the above fields, with *exactly* the same names. It should have the same make, year, accidents, and miles. Here are the changes you must make:

1. Add 5,000 to the cost if the car has clocked less than 20,000 miles.

2. Subtract 5,000 if it has clocked more than 100,000 miles.
3. Reduce the price by 10,000 for every accident.

4. This problem deals with structures that represent dates.
   a. First, write a MATLAB function called `createDate` that will take in three numeric parameters. The first parameter represents the `month`, the second the `day`, and the third the `year`. The function should return a structure with the following fields:

      `Day:` a number
      `Month:` a 3 character string containing the first three characters of the month name
      `Year:` a number containing the year.

      For example,

      `it = createDate(3,30,2008)` should return a structure containing:

      ```
      Day: 30
      Month: Mar
      Year: 2008
      ```

   b. Write a function called `printDate` that displays a date in the form `Mar 30, 2007`
   c. Write a function `inBetween` that will take in three date structures. The function should return `true` if the second date is between the first and third dates, otherwise the function should return `false`.
   d. Write a function called `isSorted` that takes in a single parameter, an array of date structures. This function should return `true` if all the dates in the array are in a chronological order (regardless of whether they are in ascending or descending order), otherwise the function should return `false`.
   e. Write a test script that creates an array of date structures, prints out each date, and then states whether or not the dates are in order.

Hints

- It might help to add a field to the date class.
- The third date does not have to be chronologically later than the first date.

5. Your university has added a new award for students who were "almost there" last semester and just missed getting into the Dean's List. Write a function called `almost` that consumes an array of student structures, and produces an array of names of those who have a semester GPA between 2.9 and 2.99 (inclusive). The student structure has the following fields:

```
Name - string (e.g., 'George P. Burdell')
Semester_GPA - decimal number (e.g., 2.97)
Cumulative_GPA - decimal number (e.g., 3.01)
```

6. The MATLAB language has the built-in ability to perform mathematical operations on complex numbers. However, there are times when it is useful to treat complex numbers as a structure. Write a set of functions with the following capability and a script to verify that they work correctly:

```
cmplx = makeComplex(real, imag)
res = cmplxAdd( cmpxa, cmpxb )
res = cmplxMult( cmpxa, cmpxb )
```

7. In terms of atomic physics, every electron has four numbers associated with it, called the quantum numbers. These are `'principal'` (energy), `'azimuthal'` (angular momentum), `'magnetic'` (orientation of angular momentum), and `'spin'` (particle spin) quantum numbers. Wolfgang Pauli hypothesized (correctly) that no two electrons in an atom can have the same set of four quantum numbers; that is, if the `Principal`, `Azimuthal`, and `Magnetic` numbers are the same for two electrons, then it is necessary for the electrons to have different `Spin` numbers.

   You need to write a function called `spinSwitch` that takes in two structures and returns both structures. Each structure represents an electron in a hydrogen atom and has the following fields:

```
principal (this is always > 0)
azimuthal (a number)
magnetic (a number)
spin (a string with value 'up' or 'down')
```

   Your function will compare the values in the two structures and check if they all have the same values for the four fields. If true, you are required to switch the spin of the second structure. You also have to add a field called “energy” to both structures. The value stored in this field must be (–2.18*(10^18))/(n^2), where n is the value of the principal quantum number for that electron. You have to return both the structures with the energy field added to both, so that the one with the higher energy is first. If the energies are equal, return the one with the ‘up’ spin first. If both have the same spin and the same energy, the order does not matter.