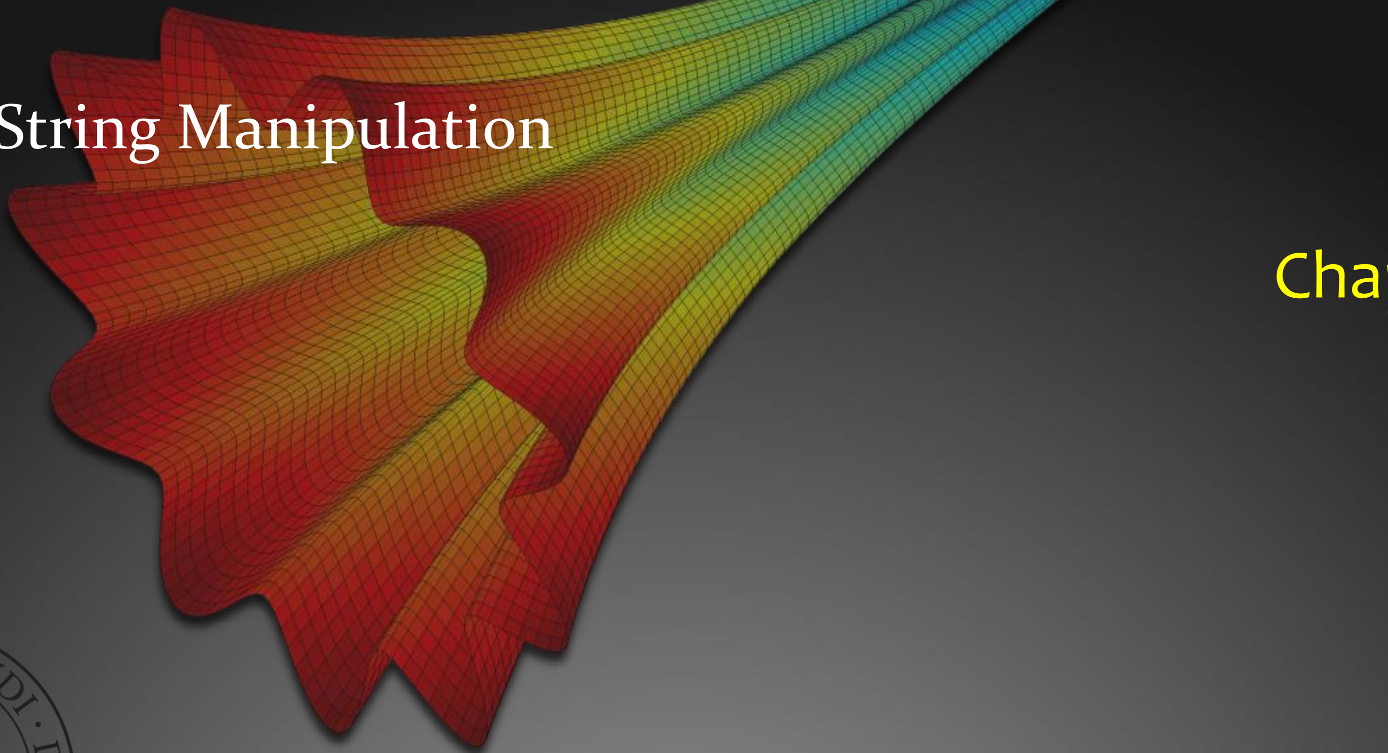


String Manipulation

Chapter 7

Linguaggio Programmazione Matlab-Simulink (2018/2019)

Strings: Terminology

- A ***string*** in MATLAB consists of any number of characters and is contained in single quotes
- strings are vectors in which every element is a single character
- A ***substring*** is a subset or part of a string
- ***Characters*** include letters of the alphabet, digits, punctuation marks, white space, and control characters
  - ***Control characters*** are characters that cannot be printed, but accomplish a task (such as a backspace or tab)
  - ***White space characters*** include the space, tab, newline, and carriage return
  - ***Leading blanks*** are blank spaces at the beginning of a string,
  - ***Trailing blanks*** are blank spaces at the end of a string
- Empty string is a string with length `0`, e.g. `''`

String Variables

- String variables can be created using
  - assignment statements
  - input function (with ‘s’ as the second argument)
- Since strings are vectors of characters, many built-in functions and operators that we’ve seen already work with strings as well as numbers – e.g., **length** to get the length of a string, or the transpose operator
- You can also index into a string variable to get individual characters or to get subsets of strings, or in other words, substrings

String Concatenation

- There are several ways to ***concatenate***, or join, strings
- To horizontally concatenate (creates one long string):
  - Using [ ], e.g.

    ```
    >> ['hello'   'there']
    ans =
    hellothere
    ```

  - Using **strcat**, e.g. strcat(‘hello’ , ‘there’)

    ```
    >> strcat('hello', 'there')
    ans =
    hellothere
    ```

  - There is a difference: if there are leading blanks, using [] will retain them whereas **strcat** will not

Vertical Concatenation

- Vertically concatenating strings creates a column vector of strings, which is basically a character matrix (a matrix in which every element is a single character)
- There are 2 ways to do this:
  - Using [ ] and separating with semicolons
  - Using **char**
- Since all rows in a matrix must have the same number of characters, shorter strings must be padded with blank spaces so that all strings are the same length ; the built-in function **char** will do that automatically

Character Matrices

- Both [ ] and char can be used to create a matrix in which every row has a string:

```
>> cmat = ['Hello';'Hi   '; 'Ciao '];
>> cmat = char('Hello', 'Hi', 'Ciao’);
```

- Both of these will create a matrix *cmat*:

H	e	l	l	o
H	i			
C	i	a	o	

- Shorter strings are padded with blanks, e.g.

```
cmat(2,:) is 'Hi   '
```

The **sprintf** function

- **sprintf** works just like **fprintf**, but instead of printing, it creates a string – so it can be used to customize the format of a string
- So, **sprintf** can be used to create customized strings to pass to other functions (e.g., **title**, **input**)

```
>> maxran = randi([1, 50]);
>> prompt = sprintf('Enter an integer from 1 to %d: ', maxran);
>> mynum = input(prompt);
Enter an integer from 1 to 46: 33
```

- Any time a string is required as an input, **sprintf** can create a customized string

String Comparisons

- **strcmp** compares two strings and returns logical 1 if they are identical or 0 if not (or not the same length)
- For strings, use this instead of the equality operator ==
- variations:
  - **strncmp** compares only the first n characters
  - **strcmpi** ignores case (upper or lower)
  - **strncmpi** compares n characters, ignoring case

Find and replace functions

- **strfind(string, substring)**: finds all occurrences of the substring within the string; returns a vector of the indices of the beginning of the strings, or an empty vector if the substring is not found
- **strrep(string, oldsubstring, newsubstring)**: finds all occurrences of the old substring within the string, and replaces with the new substring
    - the old and new substrings can be different lengths

The **strtok** function

- The **strtok** function takes a string and breaks it into two pieces and returns both strings
  - It looks for a *delimiter* (by default a blank space) and returns a *token* which is the beginning of the string up to the delimiter, and also the rest of the string, including the delimiter
  - A second argument can be passed for the delimiter
  - So – no characters are lost; all characters from the original string are returned in the two output strings
  - Since the function returns two strings, the call to **strtok** should be in an assignment statement with two variables on the left to store the two strings

Examples of **strtok**

```
>> mystring = 'Isle of Skye';
>> [first, rest] = strtok(mystring)
first =
Isle
rest =
 of Skye
>> length(rest)
ans =
     8
>> [f, r] = strtok(rest, 'y')
f =
 of Sk
r =
ye
```

The **eval** function

- The **eval** function evaluates a string as a function call or a statement
- Usually used when the contents of the string are not known ahead of time; e.g., the user enters part of it and then a customized string is created
- For example:

```
>> x = 1:5;
>> fn = input('Enter a function name: ', 's');
Enter a function name: cos
>> eval(strcat(fn, '(x)'))
ans =
    0.5403   -0.4161   -0.9900   -0.6536    0.2837
```

eval example

This is a very common application: a series of experiments has been run, resulting in files with the same name except for consecutive integers at the end of the name. We will write a **for** loop that will load files named ‘file1.dat’, ‘file2.dat’, … ‘file5.dat’ (assuming that they exist)

```
for i = 1:5
    eval(sprintf('load file%d.dat',i))
end
```

"is" & String/Number Functions

- "is" functions for strings:
  - **isletter** true if the input argument is a letter of the alphabet
  - **isspace** true if the input argument is a white space character
  - **ischar** true if the input argument is a string
  - **isstrprop** determines whether the characters in a string are in a category specified by second argument, e.g. 'alphanumeric'
- Converting from strings to numbers and vice versa:
  - **int2str** converts from an integer to a string storing the integer
  - **num2str** converts a real number to a string containing the number
  - **str2num** (and **str2double**) converts from a string containing number(s) to a number array
  - (Note: different from converting to/from ASCII equivalents)

Common Pitfalls

- Trying to use == to compare strings for equality, instead of the **strcmp** function (and its variations)
- Confusing **sprintf** and **fprintf**. The syntax is the same, but **sprintf** creates a string whereas **fprintf** prints
- Trying to create a vector of strings with varying lengths (the easiest way is to use **char** which will pad with extra blanks automatically)
- Forgetting that when using **strtok**, the second argument returned (the "rest" of the string) contains the delimiter.

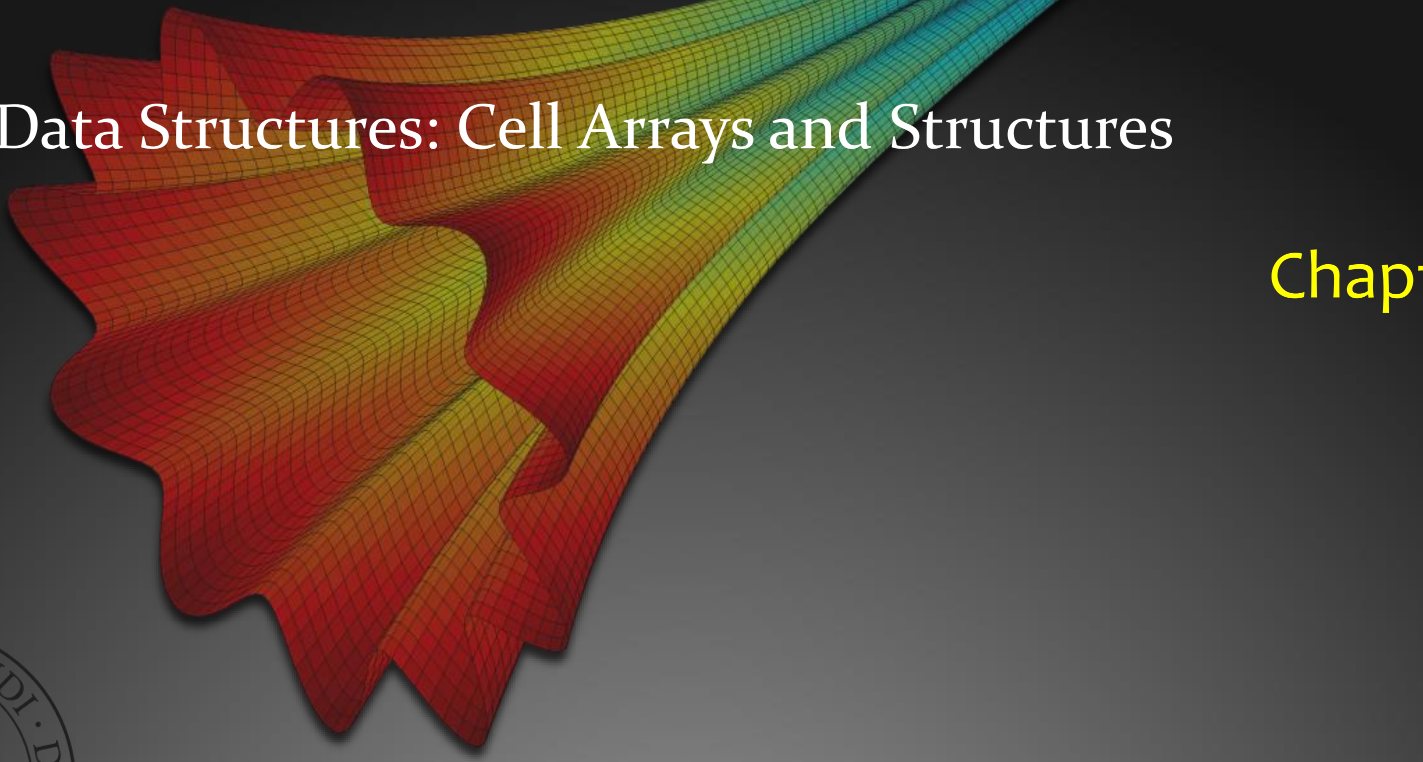
Programming Style Guidelines

- Trim trailing blanks from strings that have been stored in matrices before using

- Make sure the correct string comparison function is used; for example, **strcmpi** if ignoring case is desired

Exercises

- Prompt the user for a string. Print the length of the string and also the first and last characters in the string. Make sure that this works regardless of what the user enters.
- In a loop, create and print strings with file names “file1.dat”, “file2.dat”, and so on for file numbers 1 through 5.
- Create an *x* vector. Prompt the user for ‘sin’, ‘cos’, or ‘tan’ and create a string with that function of *x* (e.g., ‘sin(x)’ or ‘cos(x)’). Use **eval** to create a *y* vector using the specified function.



Data Structures: Cell Arrays and Structures

Chapter 8

Linguaggio Programmazione Matlab-Simulink (2018/2019)

Cell Arrays

- A ***cell array*** is a type of data structure that can store different types of values in its elements
- A cell array could be a vector (row or column) or a matrix
- It is an array, so indices are used to refer to the elements
- One great application of cell arrays: storing strings of different lengths

Creating Cell Arrays

- The syntax used to create a cell array is curly braces { } instead of [ ]
- The direct method is to put values in the row(s) separated by commas or spaces, and to separate the rows with semicolons (so, same as other arrays) – the difference is using { } instead of [ ]
- The **cell** function can also be used to preallocate by passing the dimensions of the cell array, e.g.

```
cell(4,2)
```

Referring to Cell Array Elements

- The elements in cell arrays are cells
- There are two methods of referring to parts of cell arrays:
  - you can refer to the cells; this is called cell indexing and parentheses are used
  - you can refer to the contents of the cells; this is called content indexing and curly braces are used
- For example:

```
>> ca = {2:4, 'hello'};
>> ca(1)
ans =
    [1x3 double]
>> ca{1}
ans =
     2     3     4
```

Cell Array Functions

- the **celldisp** function displays the contents of all elements of a cell array
- **cellplot** puts a graphical display in a Figure Window (but it just shows cells, not their contents)
- to convert from a character matrix to a cell array of strings: **cellstr**
- **iscellstr** will return logical true if a cell array is a cell array of all strings

Functions **strjoin** and **strsplit**

- Introduced in R2013a
- **strjoin** concatenates all strings from a cell array into one string separated by a delimiter (space by default but others can be specified)
- **strsplit** splits a string into elements in a cell array using a blank space as the default delimiter (or another specified)

Structure Variables

- Structures store values of different types, in ***fields***
- Fields are given names; they are referred to as

  structurename.fieldname using the ***dot operator***
- Structure variables can be initialized using the **struct** function, which takes pairs of arguments (field name as a string followed by the value for that field)
- To print, **disp** will display all fields; **fprintf** can only print individual fields

Struct Example

```
>> subjvar = struct('SubjNo',123,'Height',62.5);
>> subjvar.Height
ans =
   62.5000
>> disp(subjvar)
    SubjNo: 123
    Height: 62.5000
>> fprintf('The subject # is %d\n',...
                    subjvar.SubjNo)
The subject # is 123
```

Cell Arrays vs. Structs

- Cell arrays are arrays, so they are indexed
  - That means that you can loop though the elements in a cell array – or have MATLAB do that for you by using vectorized code
- Structs are not indexed, so you can not loop
  - However, the field names are mnemonic so it is more clear what is being stored in a struct
- For example:

  variable{1} vs. variable.weight: which is more mnemonic?

Structure Functions

- the function **rmfield** removes a field but doesn’t alter the variable
- the function **isstruct** will return logical 1 if the argument is a structure variable
- the function **isfield** receives a structure variable and a string and will return logical 1 if the string is the name of a field within the structure
- the **fieldnames** function receives a structure variable and returns the names of all of its fields as a cell array

Vector of Structures

- A database of information can be stored in MATLAB in a vector of stuctures; a vector in which every element is a structure
- For example, for a medical experiment information on subjects might include a subject #, the person's height, and the person's weight
- Every structure would store 3 fields: the subject #, height, and weight
- The structures would be stored together in one vector so that you could loop through them to perform the same operation on every subject – or vectorize the code

Example

```
>> subjvar(2) = struct('SubjNo', 123, 'Height',...
        62.5, 'Weight', 133.3);
>> subjvar(1) = struct('SubjNo', 345, 'Height',...
        77.7, 'Weight', 202.5);
```

- This creates a vector of 2 structures
- The second is created first to preallocate to 2 elements
- A set of fields can be created, e.g.

```
>> [subjvar.Weight]
ans =
   202.5000   133.3000
```

Example Problem

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’

We will write general statements (using the programming method) that will print the item number and code fields of each structure in a nice format

```
for i = 1:length(packages)
   fprintf('Item %d has a code of %c\n', ...
       packages(i).item_no, packages(i).code)
end
```

Nested Structures

- A nested structure is a structure in which at least one field is another structure
- To refer to the "inner" structure, the dot operator would have to be used twice
  - e.g. structurename.innerstruct.fieldname
- To create a nested structure, calls to the **struct** function can be nested

Nested struct example

- The following creates a structure for a contact that includes the person's name and phone extension. The name is a struct itself that separately stores the first and last name.
- The calls to **struct** are nested

```
>> contactinfo = struct(...
        'cname', struct('last', 'Smith', 'first', 'Abe'),...
        'phoneExt', '3456');
>> contactinfo.cname.last
ans =
Smith
```

Sorting

- Sorting is the process of putting a list in order; either ***ascending*** (lowest to highest) or ***descending*** (highest to lowest)
- MATLAB has built-in sort functions
- there are many different sort algorithms
- One strategy is to leave the original list, and create a new list with all of the same values but in sorted order – another is to sort the original list in place
- The *selection sort* will be used as an example of sorting a vector in ascending order in place

Selection sort algorithm

- Put the next smallest number in each element by looping through the elements from the first through the next-to-last (the last will then automatically be the largest value)
- For each element i:
  - find the index of the smallest value
    - start by saying the top element is the smallest so far (what is needed is to store its index)
    - loop through the rest of the vector to find the smallest
  - put the smallest in element i by exchanging the value in element i with the element in which the smallest was found

Selection Sort Function

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

% Loop through the elements in the vector to end-1
for i = 1:length(vec)-1
    indlow = i; % stores the index of the lowest
    %Select the lowest number 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
```

Built-in **sort** Function

- The **sort** function will sort a vector in ascending (default) or descending order:

```
>> vec = randi([1, 20],1,7)
vec =
   17    3    9   19   16   20   14
>> sort(vec)
ans =
    3    9   14   16   17   19   20
>> sort(vec, 'descend')
ans =
   20   19   17   16   14    9    3
```

- For a matrix, each individual column would be sorted
- sort(mat,2) sorts on rows instead of columns

Sorting Vector of Structs

- For a vector of structures, what would it mean to sort? Based on what? Which field?
- For example, for the vector "parts", it would make sense to sort based on any of the fields (code, quantity, or weight)
- The sorting would be done based on the field e.g. parts(i).code
- The entire stuctures would be exchanged

parts

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

Selection Sort for Vector of structs

- If the vector of structs is called “vec” instead of “parts”, what would change in this function to sort based on the code field?

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

% Loop through the elements in the vector to end-1
for i = 1:length(vec)-1
    indlow = i; % stores the index of the lowest
    %Select the lowest number in the rest of the vector
    for j = i+1 : length(vec)
        if ~~vec(j) < vec(indlow)~~   vec(j).code < vec(indlow).code
            indlow = j;
        end
    end
    % Exchange elements
    temp = vec(i);
    vec(i) = vec(indlow);
    vec(indlow) = temp;
end
outv = vec;
end
```

Sorting Strings

- To sort a cell array of strings alphabetically, use **sort**:

```
>> sciences = {'Physics', 'Biology', 'Chemistry'};
>> sort(sciences)
ans =
    'Biology'    'Chemistry'    'Physics’
```

- For a character matrix, however, this will not work because **sort** will just sort every individual column:

```
>> scichar = char(sciences)
scichar =
Physics
Biology
Chemistry
>> sort(scichar)
ans =
Bhelics
Chomigt
Piysosyry
```

The function **sortrows**

- The function **sortrows** will sort the rows within a column vector, so for strings this will yield an alphabetical sort:

```
>> sciences = {'Physics', 'Biology', 'Chemistry'};
>> scichar = char(sciences);
>> sortrows(scichar)
ans =

Biology
Chemistry
Physics
```

- This works for numbers, also

Indexing

- Rather than sorting the entire vector every time on a particular field, it is frequently more efficient to instead create index vectors based on the different fields
- Index vectors, which we have seen already, give the order in which the vector should be traversed
    - e.g. vec([3 1 2]) says "go through the vector in this order: the third element, then the first, then the second"

Indexing into a Vector of structs

- For the vector of structures "parts", the index vectors shown give the order in which to go through the vector based on the code field, quantity, and for the weight:

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	qi	wi
1	3	1	4
2	1	3	2
3	4	4	3
4	2	2	1

E.g.: [code_sorted, ci] = sort([parts.code])

Using an index vector

- To use this (e.g. to iterate through the vector in order of the code field by using the code index vector ci):

```
for i = 1:length(parts)
    do something with parts(ci(i))
end
```

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

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 to index into a cell array using parentheses or curly braces or referring to a field of a structure using the dot operator
- Thinking that you can index into a structure
- 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 the same type and represent in some sense the same thing.
- Use cell arrays or structures when the values are logically related but not the same type nor the same thing.
- Use cell arrays rather than character matrices when storing strings of different lengths
- 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, **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.

Exercises

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.

Exercises

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.

Exercises

Practice 8.3

- Modify the code from the preceding Quick Question to use **sprintf**.

Exercises

Practice 8.4

- 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
```