# Structures

*Structures* are MATLAB arrays with named "data containers" called *fields*. The fields of a structure can contain any kind of data. For example, one field might contain a text string representing a name, another might contain a scalar representing a billing amount, a third might hold a matrix of medical test results, and so on.

```
patient
 |
 |-.name ----------- 'John Doe'
 |-.billing -------- 127.00
 |_.test ----------- | 79   75   73    |
                     | 180  178  177.5 |
                     | 220  210  205   |
```

Like standard arrays, structures are inherently array oriented. A single structure is a 1-by-1 structure array, just as the value 5 is a 1-by-1 numeric array. You can build structure arrays with any valid size or shape, including multidimensional structure arrays.

---

**Note** The examples in this section focus on two-dimensional structure arrays. For examples of higher-dimension structure arrays, see "Multidimensional Arrays" on page 1-37.

---

The following list summarizes the contents of this section:


- "Building Structure Arrays" on page 2-50
- "Accessing Data in Structure Arrays" on page 2-52
- "Using Dynamic Field Names" on page 2-54
- "Finding the Size of Structure Arrays" on page 2-55
- "Adding Fields to Structures" on page 2-55
- "Deleting Fields from Structures" on page 2-55
- "Applying Functions and Operators" on page 2-56

- "Writing Functions to Operate on Structures" on page 2-56
- "Organizing Data in Structure Arrays" on page 2-58
- "Nesting Structures" on page 2-62
- "Function Summary" on page 2-64


## Building Structure Arrays

You can build structures in two ways:

- Using assignment statements
- Using the struct function

### Building Structure Arrays Using Assignment Statements

You can build a simple 1-by-1 structure array by assigning data to individual fields. MATLAB automatically builds the structure as you go along. For example, create the 1-by-1 patient structure array shown at the beginning of this section:

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

Now entering

```
patient
```

at the command line results in

```
name: 'John Doe'
billing: 127
test: [3x3 double]
```

patient is an array containing a structure with three fields. To expand the structure array, add subscripts after the structure name:

```
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68 70 68; 118 118 119; 172 170 169];
```

The patient structure array now has size [1 2]. Note that once a structure array contains more than a single element, MATLAB does not display

individual field contents when you type the array name. Instead, it shows a summary of the kind of information the structure contains:

```
patient
patient =
1x2 struct array with fields:
    name
    billing
    test
```

You can also use the `fieldnames` function to obtain this information. `fieldnames` returns a cell array of strings containing field names.

As you expand the structure, MATLAB fills in unspecified fields with empty matrices so that

- All structures in the array have the same number of fields.
- All fields have the same field names.

For example, entering `patient(3).name = 'Alan Johnson'` expands the `patient` array to size `[1 3]`. Now both `patient(3).billing` and `patient(3).test` contain empty matrices.

---

**Note** Field sizes do not have to conform for every element in an array. In the `patient` example, the `name` fields can have different lengths, the `test` fields can be arrays of different sizes, and so on.

---

### Building Structure Arrays Using the struct Function

You can preallocate an array of structures with the `struct` function. Its basic form is

```
strArray = struct('field1',val1,'field2',val2, ...)
```

where the arguments are field names and their corresponding values. A field value can be a single value, represented by any MATLAB data construct, or a cell array of values. All field values in the argument list must be of the same scale (single value or cell array).

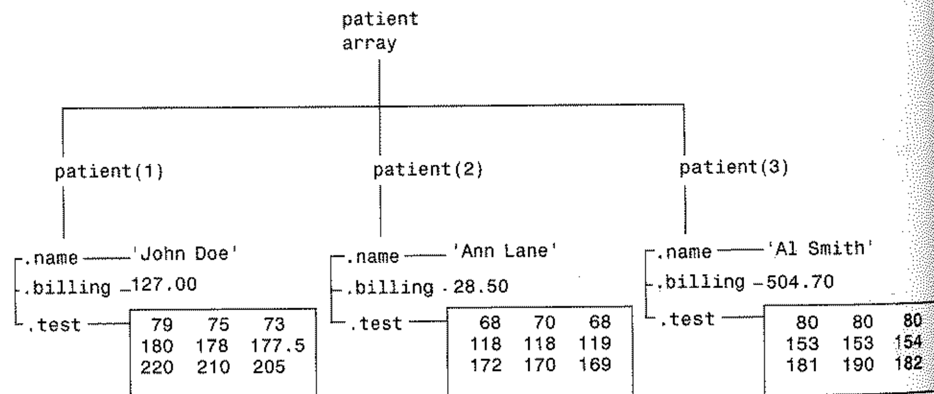
You can use different methods for preallocating structure arrays. These methods differ in the way in which the structure fields are initialized. As an

example, consider the allocation of a 1-by-3 structure array, weather, with the structure fields temp and rainfall. Three different methods for allocating such an array are shown in this table.

| Method | Syntax | Initialization |
|---|---|---|
| struct | `weather(3) = struct('temp', 72, ... 'rainfall', 0.0);` | weather(3) is initialized with the field values shown. The fields for the other structures in the array, weather(1) and weather(2), are initialized to the empty matrix. |
| struct with repmat | `weather = repmat(struct('temp', ... 72, 'rainfall', 0.0), 1, 3);` | All structures in the weather array are initialized using one set of field values. |
| struct with cell array syntax | `weather = ... struct('temp', {68, 80, 72}, ... 'rainfall', {0.2, 0.4, 0.0});` | The structures in the weather array are initialized with distinct field values specified with cell arrays. |

## Accessing Data in Structure Arrays

Using structure array indexing, you can access the value of any field or field element in a structure array. Likewise, you can assign a value to any field or field element. For the examples in this section, consider this structure array.



You can access subarrays by appending standard subscripts to a structure array name. For example, the line below results in a 1-by-2 structure array:

```
mypatients = patient(1:2)
1x2 struct array with fields:
    name
    billing
    test
```

The first structure in the `mypatients` array is the same as the first structure in the `patient` array:

```
mypatients(1)
ans =
       name: 'John Doe'
    billing: 127
       test: [3x3 double]
```

To access a field of a particular structure, include a period (.) after the structure name followed by the field name:

```
str = patient(2).name
str =
   Ann Lane
```

To access elements within fields, append the appropriate indexing mechanism to the field name. That is, if the field contains an array, use array subscripting; if the field contains a cell array, use cell array subscripting, and so on:

```
test2b = patient(3).test(2,2)
test2b =
   153
```

Use the same notations to assign values to structure fields, for example,

```
patient(3).test(2,2) = 7;
```

You can extract field values for multiple structures at a time. For example, the line below creates a 1-by-3 vector containing all of the `billing` fields:

```
bills = [patient.billing]
bills =
  127.0000   28.5000  504.7000
```

Similarly, you can create a cell array containing the `test` data for the first two structures:

```
tests = {patient(1:2).test}
tests =
    [3x3 double]    [3x3 double]
```

## Using Dynamic Field Names

The most common way to access the data in a structure is by specifying the name of the field that you want to reference. Another means of accessing structure data is to use dynamic field names. These names express the field as a variable expression that MATLAB evaluates at run-time. The dot-parentheses syntax shown here makes `expression` a dynamic field name

```
structName.(expression)
```

Index into this field using the standard MATLAB indexing syntax. For example, to evaluate *`expression`* into a field name and obtain the values of that field at columns 1 through 25 of row 7, use

```
structName.(expression)(7,1:25)
```

### Dynamic Field Names Example

The `avgscore` function shown below computes an average test score, retrieving information from the `testscores` structure using dynamic field names:

```
function avg = avgscore(testscores, student, first, last)
for k = first:last
   scores(k) = testscores.(student).week(k);
end
avg = sum(scores)/(last - first + 1);
```

You can run this function using different values for the dynamic field, `student`:

```
avgscore(testscores, 'Ann Lane', 1, 20)
ans =
   83.5000

avgscore(testscores, 'William King', 1, 20)
ans =
   92.1000
```

## Finding the Size of Structure Arrays

Use the `size` function to obtain the size of a structure array, or of any structure field. Given a structure array name as an argument, `size` returns a vector of array dimensions. Given an argument in the form `array(n).field`, the `size` function returns a vector containing the size of the field contents.

For example, for the 1-by-3 structure array `patient`, `size(patient)` returns the vector `[1 3]`. The statement `size(patient(1,2).name)` returns the length of the `name` string for element `(1,2)` of `patient`.

## Adding Fields to Structures

You can add a field to every structure in an array by adding the field to a single structure. For example, to add a social security number field to the `patient` array, use an assignment like

```
patient(2).ssn = '000-00-0000';
```

Now `patient(2).ssn` has the assigned value. Every other structure in the array also has the `ssn` field, but these fields contain the empty matrix until you explicitly assign a value to them.

To add new fields to a structure, specifying the names for these fields at run-time, see the section on "Using Dynamic Field Names" on page 2-54.

## Deleting Fields from Structures

You can remove a given field from every structure within a structure array using the `rmfield` function. Its most basic form is

```
struc2 = rmfield(array, 'field')
```

where `array` is a structure array and `'field'` is the name of a field to remove from it. To remove the `name` field from the `patient` array, for example, enter

```
patient = rmfield(patient, 'name');
```

## Applying Functions and Operators

Operate on fields and field elements the same way you operate on any other MATLAB array. Use indexing to access the data on which to operate. For example, this statement finds the mean across the rows of the `test` array in `patient(2)`:

```
mean((patient(2).test)');
```

There are sometimes multiple ways to apply functions or operators across fields in a structure array. One way to add all the `billing` fields in the `patient` array is

```
total = 0;
for k = 1:length(patient)
      total = total + patient(k).billing;
end
```

To simplify operations like this, MATLAB enables you to operate on all like-named fields in a structure array. Simply enclose the `array.field` expression in square brackets within the function call. For example, you can sum all the `billing` fields in the `patient` array using

```
total = sum ([patient.billing]);
```

This is equivalent to using the comma-separated list:

```
total = sum ([patient(1).billing, patient(2).billing, ...]);
```

This syntax is most useful in cases where the operand field is a scalar field:

## Writing Functions to Operate on Structures

You can write functions that work on structures with specific field architectures. Such functions can access structure fields and elements for processing.

---

**Note** When writing M-file functions to operate on structures, you must perform your own error checking. That is, you must ensure that the code checks for the expected fields.

---

As an example, consider a collection of data that describes measurements, at different times, of the levels of various toxins in a water source. The data consists of fifteen separate observations, where each observation contains three separate measurements.

You can organize this data into an array of 15 structures, where each structure has three fields, one for each of the three measurements taken.

The function concen, shown below, operates on an array of structures with specific characteristics. Its arguments must contain the fields lead, mercury, and chromium:

```
function [r1, r2] = concen(toxtest);
% Create two vectors. r1 contains the ratio of mercury to lead
% at each observation. r2 contains the ratio of lead to chromium.
r1 = [toxtest.mercury] ./ [toxtest.lead];
r2 = [toxtest.lead] ./ [toxtest.chromium];

% Plot the concentrations of lead, mercury, and chromium
% on the same plot, using different colors for each.
lead = [toxtest.lead];
mercury = [toxtest.mercury];
chromium = [toxtest.chromium];

plot(lead, 'r');   hold on
plot(mercury, 'b')
plot(chromium, 'y');   hold off
```

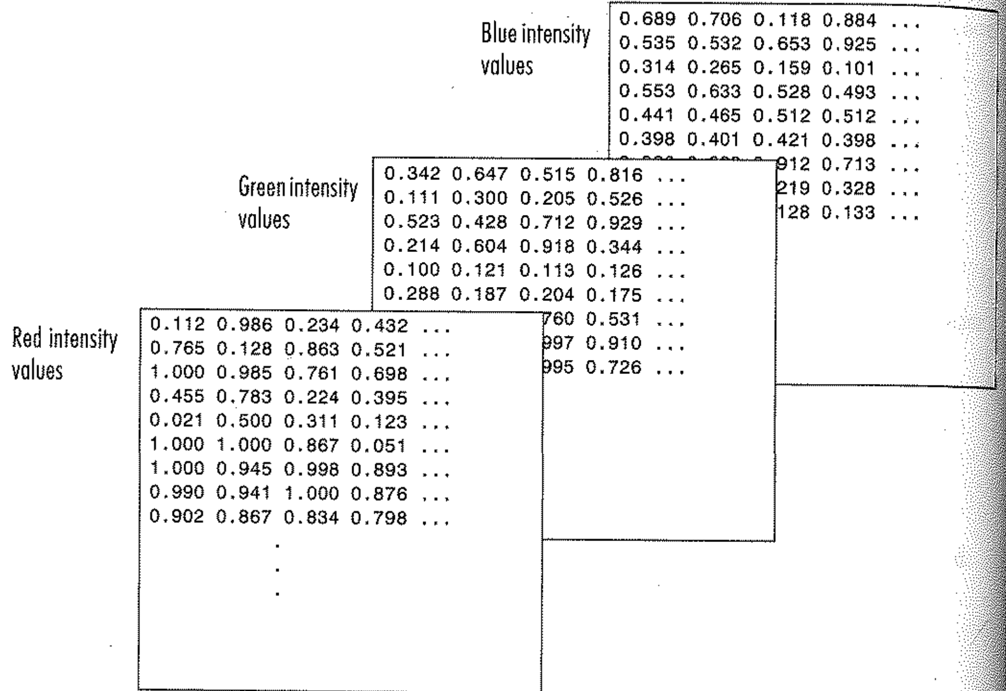
Try this function with a sample structure array like test:

```
test(1).lead = .007;
test(2).lead = .031;
test(3).lead = .019;

test(1).mercury = .0021;
test(2).mercury = .0009;
test(3).mercury = .0013;

test(1).chromium = .025;
test(2).chromium = .017;
test(3).chromium = .10;
```

## Organizing Data in Structure Arrays

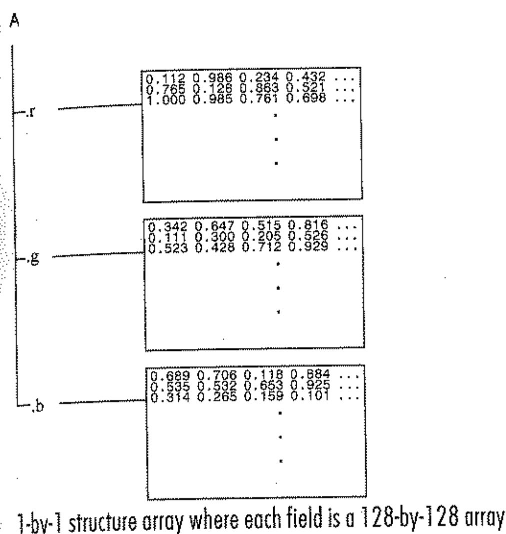
The key to organizing structure arrays is to decide how you want to access subsets of the information. This, in turn, determines how you build the array that holds the structures, and how you break up the structure fields.

For example, consider a 128-by-128 RGB image stored in three separate arrays; RED, GREEN, and BLUE.



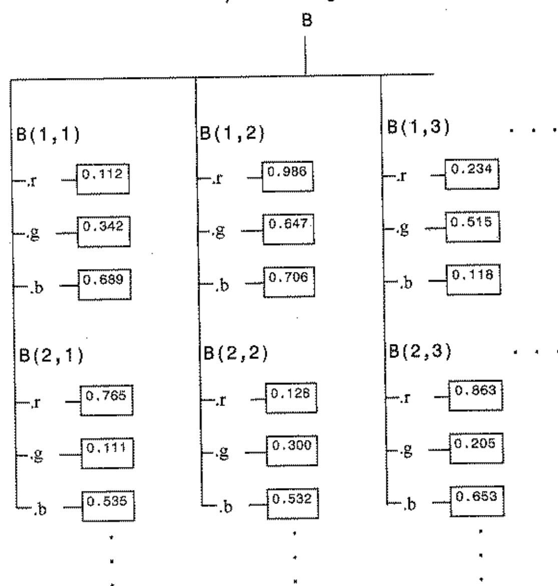
There are at least two ways you can organize such data into a structure array.

Plane organization




1-by-1 structure array where each field is a 128-by-128 array

Element-by-element organization




128-by-128 structure array where each field is a single data element

## Plane Organization

In the plane organization, shown to the left in the figure above, each field of the structure is an entire plane of the image. You can create this structure using

```
A.r = RED;
A.g = GREEN;
A.b = BLUE;
```

This approach allows you to easily extract entire image planes for display, filtering, or other tasks that work on the entire image at once. To access the entire red plane, for example, use

```
redPlane = A.r;
```

Plane organization has the additional advantage of being extensible to multiple images in this case. If you have a number of images, you can store them as `A(2)`, `A(3)`, and so on, each containing an entire image.

The disadvantage of plane organization is evident when you need to access subsets of the planes. To access a subimage, for example, you need to access each field separately:

```
redSub = A.r(2:12,13:30);
greenSub = A.g(2:12,13:30);
blueSub = A.b(2:12,13:30);
```

### Element-by-Element Organization

The element-by-element organization, shown to the right in the figure above, has the advantage of allowing easy access to subsets of data. To set up the data in this organization, use

```
for m = 1:size(RED,1)
   for n = 1:size(RED,2)
      B(m,n).r = RED(m,n);
      B(m,n).g = GREEN(m,n);
      B(m,n).b = BLUE(m,n);
   end
end
```

With element-by-element organization, you can access a subset of data with a single statement:

```
Bsub = B(1:10,1:10);
```

To access an entire plane of the image using the element-by-element method, however, requires a loop:

```
redPlane = zeros(128, 128);
for k = 1:(128 * 128)
    redPlane(k) = B(k).r;
end
```

Element-by-element organization is not the best structure array choice for most image processing applications; however, it can be the best for other applications wherein you will routinely need to access corresponding subsets of structure fields. The example in the following section demonstrates this type of application.

## Example — A Simple Database

Consider organizing a simple database.

A Plane organization

.name — 'Ann Jones' ... 'Dan Smith' ... 'Kim Lee' ...

.address — '80 Park St.' ... '5 Lake Ave.' ... '116 Elm St.' ...

.amount — 12.50 ... 81.29 ... 30.00 ...

```
A.name = strvcat('Ann Jones', ...
   'Dan Smith',...);
A.address = strvcat('80 Park St.', ...
   '5 Lake Ave.',...);
A.amount = [12.5; 81.29; 30; ...];
```

B Element-by-element organization

B(1) .name — 'Ann Jones' .address — '80 Park St.' .amount — 12.50

B(2) .name — 'Dan Smith' .address — '5 Lake Ave.' .amount — 81.29

B(3) .name — 'Kim Lee' .address — '116 Elm St.' .amount — 30.00

. . .

```
B(1).name = 'Ann Jones';
B(1).address = '80 Park St.';
B(1).amount = 12.5;

B(2).name = 'Dan Smith';
B(2).address = '5 Lake Ave.';
B(2).amount = 81.29;
.
.
.
```

Each of the possible organizations has advantages depending on how you want to access the data:

- Plane organization makes it easier to operate on all field values at once. For example, to find the average of all the values in the amount field,
  - Using plane organization

    ```
    avg = mean(A.amount);
    ```

  - Using element-by-element organization

    ```
    avg = mean([B.amount]);
    ```

- Element-by-element organization makes it easier to access all the information related to a single client. Consider an M-file, `client.m`, which displays the name and address of a given client on screen.

  Using plane organization, pass individual fields.

  ```
  function client(name,address)
  disp(name)
  disp(address)
  ```

  To call the `client` function,

  ```
  client(A.name(2,:),A.address(2,:))
  ```

  Using element-by-element organization, pass an entire structure.

  ```
  function client(B)
  disp(B)
  ```

  To call the `client` function,

  ```
  client(B(2))
  ```

- Element-by-element organization makes it easier to expand the string array fields. If you do not know the maximum string length ahead of time for plane organization, you may need to frequently recreate the `name` or `address` field to accommodate longer strings.

Typically, your data does not dictate the organization scheme you choose. Rather, you must consider how you want to access and operate on the data.

## Nesting Structures

A structure field can contain another structure, or even an array of structures. Once you have created a structure, you can use the `struct` function or direct assignment statements to nest structures within existing structure fields.

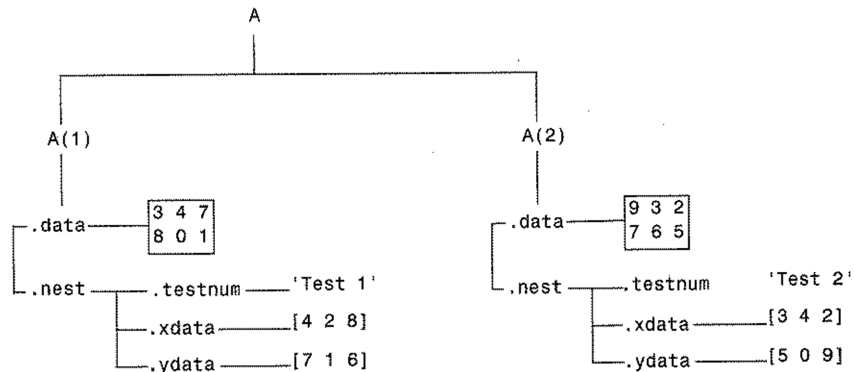
### Building Nested Structures with the struct Function

To build nested structures, you can nest calls to the `struct` function. For example, create a 1-by-1 structure array:

```
A = struct('data', [3 4 7; 8 0 1], 'nest',...
    struct('testnum', 'Test 1', 'xdata', [4 2 8],...
    'ydata', [7 1 6]));
```

You can build nested structure arrays using direct assignment statements. These statements add a second element to the array:

```
A(2).data = [9 3 2; 7 6 5];
A(2).nest.testnum = 'Test 2';
A(2).nest.xdata = [3 4 2];
A(2).nest.ydata = [5 0 9];
```




### Indexing Nested Structures

To index nested structures, append nested field names using dot notation. The first text string in the indexing expression identifies the structure array, and subsequent expressions access field names that contain other structures.

For example, the array A created earlier has three levels of nesting:

- To access the nested structure inside A(1), use A(1).nest.
- To access the xdata field in the nested structure in A(2), use A(2).nest.xdata.
- To access element 2 of the ydata field in A(1), use A(1).nest.ydata(2).

## Function Summary

This table describes the MATLAB functions for working with structures.

| Function | Description |
|---|---|
| `deal` | Deal inputs to outputs. |
| `fieldnames` | Get structure field names. |
| `isfield` | Return `true` if the field is in a structure array. |
| `isstruct` | Return `true` for structures. |
| `rmfield` | Remove a structure field. |
| `struct` | Create or convert to a structure array. |
| `struct2cell` | Convert a structure array into a cell array. |