Title

stata.com

struct — Structures

Syntax Description Remarks and examples Reference Also see

Syntax

```
struct structname {
    declaration(s)
}
```

such as

```
struct mystruct {
      real scalar    n1, n2
      real matrix    X
}
```

Description

A structure contains a set of variables tied together under one name.

Structures may not be used interactively. They are used in programming.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Introduction

Here is an overview of the use of structures:

```
struct mystruct {
            real scalar    n1, n2
            real matrix    X
}

function myfunc()
{
        struct mystruct scalar e

        ...
        e.n1 = ...
        e.n2 = ...
        e.X  = ...
        ...
        ... mysubroutine(e, ...)
        ...
}

function mysubroutine(struct mystruct scalar x, ...)
{
        struct mystruct scalar   y
        ...
        ... x.n1 ... x.n2 ... x.X ...
        ...
        y = mysubfcn(x)
        ...
        ... y.n1 ... y.n2 ... y.X ...
        ... x.n1 ... x.n2 ... x.X ...
        ...
}

struct mystruct scalar mysubfcn(struct mystruct scalar x)
{
        struct mystruct scalar   result

        result = x
        ... result.n1 ... result.n2 ... result.X ...
        return(result)
}
```

Note the following:

1. We first defined the structure. Definition does not create variables; definition defines what we mean when we refer to a `struct mystruct` in the future. This definition is done outside and separately from the definition of the functions that will use it. The structure is defined before the functions.

2. In `myfunc()`, we declared that variable `e` is a `struct mystruct scalar`. We then used variables `e.n1`, `e.n2`, and `e.X` just as we would use any other variable. In the call to `mysubroutine()`, however, we passed the entire `e` structure.

3. In `mysubroutine()`, we declared that the first argument received is a `struct mystruct scalar`. We chose to call it `x` rather than `e` to emphasize that names are not important. `y` is also a `struct mystruct scalar`.

4. `mysubfcn()` not only accepts a `struct mystruct` as an argument but also returns a `struct mystruct`. One of the best uses of structures is as a way to return multiple, related values.

   The line `result=x` copied all the values in the structure; we did not need to code `result.n1=x.n1`, `result.n2=x.n2`, and `result.X=x.X`.

Structures and functions must have different names

You define structures much as you define functions, at the colon prompt, with the definition enclosed in braces:

```
: struct twopart {
>         real scalar    n1, n2
> }
: function setuphistory()
> {
>         ...
> }
```

Structures and functions may not have the same names. If you call a structure `twopart`, then you cannot have a function named `twopart()`, and vice versa.

Structure variables must be explicitly declared

Declarations are usually optional in Mata. You can code

```
real matrix swaprows(real matrix A, real scalar i1, real scalar i2)
{
      real matrix     B
      real rowvector  v

      B = A
      v = B[i1, .]
      B[i1, .] = B[i2, .]
      B[i2, .] = v
      return(B)
}
```

or you can code

```
function swaprows(A, i1, i2)
{
      B = A
      v = B[i1, .]
      B[i1, .] = B[i2, .]
      B[i2, .] = v
      return(B)
}
```

When a variable, argument, or returned value is a structure, however, you must explicitly declare it:

```
function makecalc()
{
     struct twopart scalar   t
     t.n1 = t.n2 = 0
     ...
}

function clear_twopart(struct twopart scalar t)
{
     t.n1 = t.n2 = 0
}

struct twopart scalar new_twopart()
{
     struct twopart scalar   t
     t.n1 = t.n2 = 0
     return(t)
}
```

In the functions above, we refer to variables `t.n1` and `t.n2`. The Mata compiler cannot interpret those names unless it knows the definition of the structure.

Aside: all structure references are resolved at compile time, not run time. That is, terms like `t.n1` are not stored in the compiled code and resolved during execution. Instead, the Mata compiler accesses the structure definition when it compiles your code. The compiler knows that `t.n1` refers to the first element of the structure and generates efficient code to access it.

Declare structure variables to be scalars whenever possible

In our declarations, we code things like

```
struct twopart scalar   t
```

and do not simply code

```
struct twopart          t
```

although the simpler statement would be valid.

Structure variables can be scalars, vectors, or matrices; when you do not say which, matrix is assumed.

Most uses of structures are as scalars, and the compiler will generate more efficient code if you tell it that the structures are scalars. Also, when you use structure vectors or matrices, there is an extra step you need to fill in, as described in the next section.

Vectors and matrices of structures

Just as you can have `real` scalars, vectors, or matrices, you can have structure scalars, vectors, or matrices. The following are all valid:

```
struct twopart scalar     t
struct twopart vector     t
struct twopart rowvector  t
struct twopart colvector  t
struct twopart matrix     t
```

In a `struct twopart matrix`, every element of the matrix is a separate structure. Say that the matrix were 2×3. Then you could refer to any of the following variables,

```
t[1,1].n1
t[1,2].n1
t[1,3].n1
t[2,1].n1
t[2,2].n1
t[2,3].n1
```

and similarly for `t[`i`,`j`].n2`.

If `struct twopart` also contained a matrix `X`, then

```
t[i,j].X
```

would refer to the (i,j)th matrix.

```
t[i,j].X[k,l]
```

would refer to the (k, l)th element of the (i,j)th matrix.

If `t` is to be a 2×3 matrix, you must arrange to make it 2×3. After the declaration

```
struct twopart matrix   t
```

`t` is 0×0. This result is no different from the situation where `t` is a `real matrix` and after declaration, `t` is 0×0.

Whether `t` is a `real matrix` or a `struct twopart matrix`, you allocate `t` by assignment. Let's pretend that `t` is a `real matrix`. There are three solutions to the allocation problem:

(1) `t = x`

(2) `t = somefunction(...)`

(3) `t = J(`r`, `c`, `v`)`

All three are so natural that you do not even think of them as allocation; you think of them as definition.

The situation with structures is the same.

Let's take each in turn.

1. `x` contains a 2×3 `struct twopart`. You code

   ```
   t = x
   ```

   and now `t` contains a copy of `x`. `t` is 2×3.

2. `somefunction(...)` returns a 2 × 3 `struct twopart`. You code

```
t = somefunction(...)
```

   and now `t` contains the 2 × 3 result.

3. Mata function `J(r, c, v)` returns an $r \times c$ matrix, every element of which is set to v. So pretend that variable `tpc` contains a `struct twopart scalar`. You code

```
t = J(2, 3, tpc)
```

   and now `t` is 2 × 3, every element of which is a copy of `tpc`. Here is how you might do that:

```
function ...(...)
{
        struct twopart scalar   tpc
        struct twopart matrix   t

        ...
        t = J(2, 3, tpc)
        ...
}
```

Finally, there is a fourth way to create structure vectors and matrices. When you defined

```
struct twopart {
        real scalar    n1, n2
}
```

Mata not only recorded the structure definition but also created a function named `twopart()` that returns a `struct twopart`. Thus, rather than enduring all the rigmarole of creating a matrix from a preallocated scalar, you could simply code

```
t = J(2, 3, twopart())
```

In fact, the function `twopart()` that Mata creates for you allows 0, 1, or 2 arguments:

`twopart()`	returns a 1 × 1 `struct twopart`
`twopart(r)`	returns an $r \times 1$ `struct twopart`
`twopart(r, c)`	returns an $r \times c$ `struct twopart`

so you could code

```
t = twopart(2, 3)
```

or you could code

```
t = J(2, 3, twopart())
```

and whichever you code makes no difference.

Either way, what is in `t`? Each element contains a separate `struct twopart`. In each `struct twopart`, the scalars have been set to missing (`.`, `""`, or `NULL`, as appropriate), the vectors and row vectors have been made 1 × 0, the column vectors 0 × 1, and the matrices 0 × 0.

Structures of structures

Structures may contain other structures:

```
struct twopart {
        real scalar    n1, n2
}

struct pair_of_twoparts {
        struct twopart scalar  t1, t2
}
```

If `t` were a `struct pair_of_twoparts scalar`, then the members of `t` would be

```
t.t1       a struct twopart scalar
t.t2       a struct twopart scalar
t.t1.n1    a real scalar
t.t1.n2    a real scalar
t.t2.n1    a real scalar
t.t2.n2    a real scalar
```

You may also create structures of structures of structures, structures of structures of structures of structures, and so on. You may not, however, include a structure in itself:

```
struct recursive {
        ...
        struct recursive scalar  r
        ...
}
```

Do you see how, even in the scalar case, `struct recursive` would require an infinite amount of memory to store?

Pointers to structures

What you can do is this:

```
struct recursive {
        ...
        pointer(struct recursive scalar) scalar  r
        ...
}
```

Thus, if `r` were a `struct recursive scalar`, then `*r.r` would be the next structure, or `r.r` would be NULL if there were no next structure. Immediately after allocation, `r.r` would equal `NULL`.

In this way, you may create linked lists.

Mata provides operator `->` for accessing members of pointers to structures.

Let `rec` be a `struct recursive`, and assume that `struct recursive` also had member `real scalar n`, so that `rec.n` would be `rec`'s `n` value. The value of the next structure's `n` would be `rec.r->n` (assuming `rec.r!=NULL`).

The syntax of -> is

exp1 -> *exp2*

where *exp1* evaluates to a structure pointer and *exp2* indexes the structure.

Operators and functions for use with structure members

All operators, all functions, and all features of Mata work with members of structures. That is, given

```
struct example {
        real scalar n
        real matrix X
}

function ...(...)
{
        real scalar rs
        real matrix rm
        struct example scalar ex

        ...
}
```

then ex.n and ex.X may be used anyplace rs and rm would be valid.

Operators and functions for use with entire structures

Some operators and functions can be used with entire structures, not just the structure's elements. Given

```
struct mystruct scalar    ex1, ex2, ex3, ex4
struct mystruct matrix    E, F, G
```

1. You may use == and != to test for equality:

   ```
   if (ex1==ex2) ...
   if (ex1!=ex2) ...
   ```

   Two structures are equal if their members are equal.

   In the example, struct mystruct itself contains no substructures. If it did, the definition of equality would include checking the equality of substructures, sub-substructures, etc.

   In the example, ex1 and ex2 are scalars. If they were matrices, each element would be compared, and the matrices would be equal if the corresponding elements were equal.

2. You may use :== and :!= to form pattern matrices of equality and inequality.

3. You may use the comma and backslash operators to form vectors and matrices of structures:

   ```
   ex = ex1, ex2 \ ex3, ex4
   ```

4. You may use & to obtain pointers to structures:

   ```
   ptr_to_ex1 = &ex1
   ```

5. You may use subscripting to access and copy structure members:

```
ex1 = E[1,2]
E[1,2] = ex1
F = E[2,.]
E[2,.] = F
G = E[|1,1\2,2|]
E[|1,1\2,2|] = G
```

6. You may use the rows() and cols() functions to obtain the number of rows and columns of a matrix of structures.

7. You may use eltype() and orgtype() with structures. eltype() returns struct; orgtype() returns the usual results.

8. You may use most functions that start with the letters *is*, as in isreal(), iscomplex(), isstring(), etc. These functions return 1 if true and 0 if false and with structures, usually return 0.

9. You may use swap() with structures.

Listing structures

To list the contents of a structure variable, as for debugging purposes, use function liststruct(); see [M-5] **liststruct()**.

Using the default, unassigned-expression method to list structures is not recommended, because all that is shown is a pointer value instead of the structure itself.

Use of transmorphics as passthrus

A transmorphic matrix can theoretically hold anything, so when we told you that structures had to be explicitly declared, that was not exactly right. Say that function twopart() returns a struct twopart scalar. You could code

```
x = twopart()
```

without declaring x (or declaring it transmorphic), and that would not be an error. What you could not do would be to then refer to x.n1 or x.n2, because the compiler would not know that x contains a struct twopart and so would have no way of interpreting the variable references.

This property can be put to good use in implementing handles and passthrus.

Say that you are implementing a complicated system. Just to fix ideas, we'll pretend that the system finds the maximum of user-specified functions and that the system has many bells and whistles. To track a given problem, let's assume that your system needs many variables. One variable records the method to be used. Another records whether numerical derivatives are to be used. Another records the current gradient vector. Another records the iteration count, and so on. There might be hundreds of these variables.

You bind all these variables together in one structure:

```
struct maxvariables {
        real scalar  method
        real scalar  use_numeric_d
        real vector  gradient
        real scalar  iteration
        ...
}
```

You design a system with many functions, and some functions call others, but because all the status variables are bound together in one structure, it is easy to pass the values from one function to another.

You also design a system that is easy to use. It starts by the user "opening" a problem,

```
handle = maximize_open()
```

and from that point on the user passes the handle around to the other maximize routines:

```
maximize_set_use_numeric_d(handle, 1)
maximize_set_function_to_max(handle, &myfunc())
...
maximize_maximize_my_function(handle)
```

In this way, you, the programmer of this system, can hold on to values from one call to the next, and you can change the values, too.

What you never do, however, is tell the user that the handle is a `struct maxvariables`. You just tell the user to open a problem by typing

```
handle = maximize_open()
```

and then to pass the handle returned to the other `maximize_*()` routines. If the user feels that he must explicitly declare the handle, you tell him to declare it:

```
transmorphic scalar handle
```

What is the advantage of this secrecy? You can be certain that the user never changes any of the values in your `struct maxvariables` because the compiler does not even know what they are.

Thus you have made your system more robust to user errors.

Saving compiled structure definitions

You save compiled structure definitions just as you save compiled function definitions; see [M-3] **mata mosave** and [M-3] **mata mlib**.

When you define a structure, such as `twopart`,

```
struct twopart {
        real scalar    n1, n2
}
```

that also creates a function, `twopart()`, that creates instances of the structure.

Saving `twopart()` in a `.mo` file, or in a `.mlib` library, saves the compiled definition as well. Once `twopart()` has been saved, you may write future programs without bothering to define `struct twopart`. The definition will be automatically found.

Saving structure variables

Variables containing structures may be saved on disk just as you would save any other variable. No special action is required. See [M-3] **mata matsave** and see the function `fputmatrix()` in [M-5] **fopen( )**. `mata matsave` and `fputmatrix()` both work with structure variables, although their entries do not mention them.

Reference

Gould, W. W. 2007. Mata Matters: Structures. *Stata Journal* 7: 556–570.

Also see

[M-2] **declarations** — Declarations and types

[M-2] **intro** — Language definition