Description Syntax Remarks and examples Also see

Description

The type and the use of declarations are explained. Also discussed is the calling convention (functions are called by address, not by value, and so may change the caller's arguments), and the use of external globals.

Mata also has structures—the *eltype* is `struct` *name*—but these are not discussed here. For a discussion of structures, see [M-2] **struct**.

Mata also has classes—the *eltype* is `class` *name*—but these are not discussed here. For a discussion of classes, see [M-2] **class**.

Declarations are optional but, for careful work, their use is recommended.

Syntax

declaration$_1$ *fcnname*`(`*declaration*$_2$`)`
`{`
    declaration$_3$
    ...
`}`

such as

```
real matrix myfunction(real matrix X, real scalar i)
{
   real scalar     j, k
   real vector     v

   ...
}
```

declaration$_1$ is one of

`function`
type [`function`]
`void` [`function`]

declaration$_2$ is

[*type*] *argname* [, [*type*] *argname* [, ...]]

where *argname* is the name you wish to assign to the argument.

declaration$_3$ are lines of the form of either of

type *varname* [, *varname* [, ...]]
`external` [*type*] *varname* [, *varname* [, ...]]

type is defined as one of

eltype orgtype	such as `real vector`
eltype	such as `real`
orgtype	such as `vector`

eltype and *orgtype* are each one of

eltype	*orgtype*
`transmorphic`	`matrix`
`numeric`	`vector`
`real`	`rowvector`
`complex`	`colvector`
`string`	`scalar`
`pointer`	

If *eltype* is not specified, `transmorphic` is assumed. If *orgtype* is not specified, `matrix` is assumed.

Remarks and examples

Remarks are presented under the following headings:

The purpose of declarations

Declarations occur in three places: in front of function definitions, inside the parentheses defining the function's arguments, and at the top of the body of the function, defining private variables the function will use. For instance, consider the function

```
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)
}
```

This function returns a copy of matrix `A` with rows `i1` and `i2` swapped.

There are three sets of declarations in the above function. First, there is a declaration in front of the function name:

```
real matrix swaprows(...)
{
     ...
}
```

That declaration states that this function will return a real matrix.

The second set of declarations occur inside the parentheses:

```
... swaprows(real matrix A, real scalar i1, real scalar i2)
{
     ...
}
```

Those declarations state that this function expects to receive three arguments, which we chose to call `A`, `i1`, and `i2`, and which we expect to be a real matrix, a real scalar, and a real scalar, respectively.

The third set of declarations occur at the top of the body of the function:

```
... swaprows(...)
{
     real matrix     B
     real rowvector  v

     ...
}
```

Those declarations state that we will use variables `B` and `v` inside our function and that, as a matter of fact, `B` will be a real matrix and `v` a real row vector.

We could have omitted all those declarations. Our function could have read

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

and it would have worked just fine. So why include the declarations?

1. By including the outside declaration, we announced to other programs what to expect. They can depend on `swaprows()` returning a real matrix because, when `swaprows()` is done, Mata will verify that the function really is returning a real matrix and, if it is not, abort execution.

   Without the outside declaration, anything goes. Our function could return a real scalar in one case, a complex row vector in another, and nothing at all in yet another case.

   Including the outside declaration makes debugging easier.

2. By including the argument declaration, we announced to other programmers what they are expected to pass to our function. We have made it easier to understand our function.

   We have also told Mata what to expect and, if some other program attempts to use our function incorrectly, Mata will stop execution.

   Just as in (1), we have made debugging easier.

3. By including the inside declaration, we have told Mata what variables we will need and how we will be using them. Mata can do two things with that information: first, it can make sure that we are using the variables correctly (making debugging easier again), and second, Mata can produce more efficient code (making our function run faster).

Interactively, we admit that we sometimes define functions without declarations. For more careful work, however, we include them.

Types, element types, and organizational types

When you use Mata interactively, you just willy-nilly create new variables:

```
: n = 2
: A = (1,2 \ 3,4)
: z = (sqrt(-4+0i), sqrt(4))
```

When you create a variable, you may not think about the type, but Mata does. `n` above is, to Mata, a real scalar. `A` is a real matrix. `z` is a complex row vector.

Mata thinks of the type of a variable as having two parts:

1. the type of the elements the variable contains (such as real or complex) and
2. how those elements are organized (such as a row vector or a matrix).

We call those two types the *eltype*—element type—and *orgtype*—organizational type. The *eltypes* and *orgtypes* are

eltype	*orgtype*
`transmorphic`	`matrix`
`numeric`	`vector`
`real`	`rowvector`
`complex`	`colvector`
`string`	`scalar`
`pointer`	

You may choose one of each and so describe all the types Mata understands.

Implicit declarations

When you do not declare an object, Mata behaves as if you declared it to be `transmorphic matrix`:

1. `transmorphic` means that the matrix can be `real`, `complex`, `string`, or `pointer`.
2. `matrix` means that the organization is to be $r \times c$, $r \geq 0$ and $c \geq 0$.

At one point in your function, a `transmorphic` matrix might be a `real scalar` (`real` being a special case of `transmorphic` and `scalar` being a special case of a `matrix` when $r = c = 1$), and at another point, it might be a `string colvector` (`string` being a special case of `transmorphic`, and `colvector` being a special case of a `matrix` when $c = 1$).

Consider our `swaprows()` function without declarations,

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

The result of compiling this function is just as if the function read

```
transmorphic matrix swaprows(transmorphic matrix A,
                             transmorphic matrix i1,
                             transmorphic matrix i2)
{
     transmorphic matrix     B
     transmorphic matrix     v

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

When we declare a variable, we put restrictions on it.

Element types

There are six *eltypes*, or element types:

1. `transmorphic`, which means `real`, `complex`, `string`, or `pointer`.
2. `numeric`, which means `real` or `complex`.
3. `real`, which means that the elements are real numbers, such as 1, 3, −50, and 3.14159.
4. `complex`, which means that each element is a pair of numbers, which are given the interpretation $a + bi$. `complex` is a storage type; the number stored in a `complex` might be real, such as $2 + 0i$.
5. `string`, which means the elements are strings of text. Each element may contain up to 2,147,483,647 bytes and strings may (need not) contain binary 0; that is, strings may be binary strings or text strings. Mata strings are similar to the `strL` type in Stata in that they can be very long and may contain binary 0. Mata strings, like all other strings in Stata, can contain Unicode characters and are stored in UTF-8 encoding.
6. `pointer` means the elements are pointers to (addresses of) other Mata matrices, vectors, scalars, or even functions; see [M-2] **pointers**.

Organizational types

There are five *orgtypes*, or organizational types:

1. `matrix`, which means $r \times c$, $r \geq 0$ and $c \geq 0$.
2. `vector`, which means $1 \times n$ or $n \times 1$, $n \geq 0$.
3. `rowvector`, which means $1 \times n$, $n \geq 0$.
4. `colvector`, which means $n \times 1$, $n \geq 0$.
5. `scalar`, which means 1×1.

Sharp-eyed readers will note that vectors and matrices can have zero rows or columns! See [M-2] **void** for more information.

Function declarations

Function declarations are the declarations that appear in front of the function name, such as

```
real matrix swaprows(...)
{
    ...
}
```

The syntax for what may appear there is

```
function
type [function]
void [function]
```

Something must appear in front of the name, and if you do not want to declare the type (which makes the type `transmorphic matrix`), you just put the word `function`:

```
function swaprows(...)
{
    ...
}
```

You may also declare the type and include the word `function` if you wish,

```
real matrix function swaprows(...)
{
    ...
}
```

but most programmers omit the word `function`; it makes no difference.

In addition to all the usual types, `void` is a type allowed only with functions—it states that the function returns nothing:

```
void _swaprows(real matrix A, real scalar i1, real scalar i2)
{
    real rowvector  v

    v = A[i1, .]
    A[i1, .] = A[i2, .]
    A[i2, .] = v
}
```

The function above returns nothing; it instead modifies the matrix it is passed. That might be useful to save memory, especially if every use of the original `swaprows()` was going to be

```
A = swaprows(A, i1, i2)
```

In any case, we named this new function `_swaprows()` (note the underscore), to flag the user that there is something odd and deserving caution concerning the use of this function.

`void`, that is to say, returning nothing, is also considered a special case of a `transmorphic matrix` because Mata secretly returns a 0×0 real matrix, which the caller just discards.

Argument declarations

Argument declarations are the declarations that appear inside the parentheses, such as

```
... swaprows(real matrix A, real scalar i1, real scalar i2)
{
        ...
}
```

The syntax for what may appear there is

[type] argname [, [type] argname [, ...]]

The names are required—they specify how we will refer to the argument—and the types are optional. Omit the type and `transmorphic matrix` is assumed. Specify the type, and it will be checked when your function is called. If the caller attempts to use your function incorrectly, Mata will stop the execution and complain.

The by-address calling convention

Arguments are passed to functions by address, not by value. If you change the value of an argument, you will change the caller's argument. That is what made `_swaprows()` (above) work. The caller passed us `A` and we changed it. And that is why in the original version of `swaprows()`, the first line read

```
B = A
```

we did our work on `B`, and returned `B`. We did not want to modify the caller's original matrix.

You do not ordinarily have to make copies of the caller's arguments, but you do have to be careful if you do not want to change the argument. That is why in all the official functions (with the single exception of `st_view()`—see [M-5] **st_view( )**), if a function changes the caller's argument, the function's name starts with an underscore. The reverse logic does not hold: some functions start with an underscore and do not change the caller's argument. The underscore signifies caution, and you need to read the function's documentation to find out what it is you need to be cautious about.

Variable declarations

The variable declarations are the declarations that appear at the top of the body of a function:

```
... swaprows(...)
{
        real matrix     B
        real rowvector  v

        ...
}
```

These declarations are optional. If you omit them, Mata will observe that you are using `B` and `v` in your code, and then Mata will compile your code just as if you had declared the variables to be `transmorphic matrix`, meaning that the resulting compiled code might be a little more inefficient than it could be, but that is all.

The variable declarations are optional as long as you have not `mata set matastrict on`; see [M-3] **mata set**. Some programmers believe so strongly that variables really ought to be declared that Mata provides a provision to issue an error when they forget.

In any case, these declarations—explicit or implicit—define the variables we will use. The variables we use in our function are private—it does not matter if there are other variables named B and v floating around somewhere. Private variables are created when a function is invoked and destroyed when the function ends. The variables are private but, as explained above, if we pass our variables to another function, that function may change their values. Most functions do not.

The syntax for declaring variables is

type *varname* [, *varname* [, ...]]

`external` [*type*] *varname* [, *varname* [, ...]]

`real matrix B` and `real rowvector v` match the first syntax.

Linking to external globals

The second syntax has to do with linking to global variables. When you use Mata interactively and type

```
: n = 2
```

you create a variable named `n`. That variable is global. When you code inside a function

```
... myfunction(...)
{
     external n
     ...
}
```

The `n` variable your function will use is the global variable named `n`. If your function were to examine the value of `n` right now, it would discover that it contained 2.

If the variable does not already exist, the statement `external n` will create it. Pretend that we had not previously defined `n`. If `myfunction()` were to examine the contents of `n`, it would discover that `n` is a 0×0 matrix. That is because we coded

```
external n
```

and Mata behaved as if we had coded

```
external transmorphic matrix n
```

Let's modify `myfunction()` to read:

```
... myfunction(...)
{
     external real scalar n
     ...
}
```

Let's consider the possibilities:

1. `n` does not exist. Here `external real scalar n` will create `n`—as a real scalar, of course—and set its value to missing.

   If `n` had been declared a `rowvector`, a 1×0 vector would have been created.

   If `n` had been declared a `colvector`, a 0×1 vector would have been created.

If n had been declared a vector, a 0×1 vector would have been created. Mata could just as well have created a 1×0 vector, but it creates a 0×1.

If n had been declared a matrix, a 0×0 matrix would have been created.

2. n exists, and it is a real scalar. Our function executes, using the global n.

3. n exists, and it is a real 1×1 rowvector, colvector, or matrix. The important thing is that it is 1×1; our function executes, using the global n.

4. n exists, but it is complex or string or pointer, or it is real but not 1×1. Mata issues an error message and aborts execution of our function.

Complicated systems of programs sometimes find it convenient to communicate via globals. Because globals are globals, we recommend that you give your globals long names. A good approach is to put the name of your system as a prefix:

```
... myfunction(...)
{
      external real scalar mysystem_n
      ...
}
```

For another approach to globals, see [M-5] **findexternal( )** and [M-5] **valofexternal( )**.

Also see

[M-2] **Intro** — Language definition