The Stata Journal (2007)
7, Number 4, pp. 556–570

Mata Matters: Structures

William Gould
StataCorp
College Station, TX
wgould@stata.com

Abstract. Mata is Stata's matrix language. In the Mata Matters column, we show how Mata can be used interactively to solve problems and as a programming language to add new features to Stata. Structures are the subject of this column. Structures are an advanced programming technique that can greatly simplify complicated code.

Keywords: pr0035, Mata, structures, `struct`

1 Introduction

Structures simplify complicated programs, particularly those that involve many subroutines. Using structures makes such programs easier to design, code, and modify. Structures can even make complicated programs easier to use.

A structure is an aggregate of variables under one name, such as

```
struct mystruct {
        real scalar     a
        real scalar     b
}
```

The above is called a structure definition and is shown exactly as you would enter it into Mata. It defines a new *type*, `struct mystruct`, and that new type is conceptually no different from the already existing types of `real`, `complex`, and `string`. Just as one can have `real` scalars, `real` vectors, and `real` matrices in a program, one can have `struct mystruct` scalars, `struct mystruct` vectors, and `struct mystruct` matrices. The function below has all three:

```
function myfunc(...)
{
        struct mystruct scalar  s
        struct mystruct vector  v
        struct mystruct matrix  M

        ...
}
```

Inside `myfunc()`, one accesses the components of the structures `s`, `v`, and `M` by referring to `s.a` and `s.b`, `v[`i`].a` and `v[`i`].b`, and `M[`i`,`j`].a` and `M[`i`,`j`].b`.

One can also refer to `s`, `v`, and `M` in their entirety, which is especially useful in making code more readable. In what follows, we pass the entirety of `s`, `v`, and `M` to subroutines `mysub1()`, `mysub2()`, and `mysub3()`:

```
function myfunc(...)
{
        struct mystruct scalar  s
        struct mystruct vector  v
        struct mystruct matrix  M
        ...
        real scalar             x, y, z

        ...
        x = mysub1(s)                           // <- new
        y = mysub2(v)                           // <- new
        z = mysub3(M)                           // <- new
        ...
}
```

For `mysub1()`, we did not code `x = mysub1(s.a, s.b)`; we simply coded `x = mysub1(s)` and that gave the subroutine access to everything in `s`, namely, `s.a` and `s.b`. In our example, the structure contains 2 components, but in a real programming application, `s` might contain 40. Whether 2 or 40, the code for the subroutines reads the same way:

```
real scalar mysub1(struct mystruct scalar s)
{
        ...
}

real scalar mysub2(struct mystruct vector v)
{
        ...
}

real scalar mysub3(struct mystruct matrix M)
{
        ...
}
```

If we need to refer to the components of the structure inside one of the subroutines, we do that in the same way we did in the main program, namely, by coding `s.a` and `s.b` in `mysub1()`, `v[`i`].a` and `v[`i`].b` in `mysub2()`, `M[`i`,`j`].a`, and `M[`i`,`j`].b` in `mysub3()`. As with arguments generally, we did not have to name the arguments `s`, `v`, and `M`, the same as in the main program. The component names, however, are fixed. If `mysub1()` had been declared `mysub1(struct mystruct scalar hset)`, then inside `mysub1()`, we would refer to `hset.a` and `hset.b`.

In addition to receiving entire structures as arguments, subroutines can return entire structures. Our main routine might call a subroutine `mysub4()` that returns a `struct mystruct` vector:

```
function myfunc(...)
{
        struct mystruct scalar  s
        struct mystruct vector  v
        struct mystruct matrix  M
        ...
        real scalar               x, y, z

        ...
        x = mysub1(s)
        y = mysub2(v)
        z = mysub3(M)
        ...
        v = mysub4(...)                     // <- new
        ...
}
```

In the new line `v = mysub4(...)`, `mysub4()` returns not just an entire structure, but a vector of entire structures. Subroutine `mysub4()` would be coded

```
struct mystruct vector mysub4(...)
{
        struct mystruct vector   new

        ...
        return(new)
}
```

This ability to package a collection of variables under one name can result in significant code simplification. One can write subroutines that receive a problem and return a solution, where problem and solution are each single variables consisting of many parts. The code for a particular problem might read

```
struct problem {
        ...
}

struct solution {
        ...
}

function main_routine(dep_varname, indep_varnames, touse_varname)
{
        struct problem scalar   p
        struct solution scalar  s

        p = set_up_problem(dep_varname, indep_varnames, touse_varname)
        check_assumptions(p)
        s = get_solution(p)
        display_problem_header(p)
        display_solution(s)
        post_results_to_stata(s)
}
```

Notice how easy the code is to read and therefore to modify.

2 When to use structures

One use of the structures is to implement new element types. For instance, if Mata had not included built-in type `complex`, and we found ourselves needing complex numbers, we could define

```
struct complex {
        real scalar   re, im
}
```

Because structure variables can be declared scalars, vectors, and matrices, with that single definition, we now have complex scalars, complex vectors, and complex matrices. We could now write the necessary complex-number manipulation functions, such as those for complex arithmetic, and we would be on our way to a solution. The advantage of this approach is that when we write the main part of our program, we write our code as if Mata had complex numbers all along. The only difference is that everywhere a real Mata program has `complex`, ours would have `struct complex`.

Another use of structures is to simplify the bookkeeping required in complicated problems, and here structure scalars are the most useful. The way I am using the word complicated—what distinguishes a complicated problem from a simple one—is the amount of information necessary to describe it. Say that the problem is statistical: there might be a vector y, a matrix X, another matrix of exogenous variables Z, a scalar *vcetype* that indicates how a variance matrix is to be calculated, and yet another scalar coding that indicates how y is coded. In solving this complicated problem, we will need to pass various parts of this information to various subroutines. The easy way to do that is to define one structure to hold all the information,

```
struct problemdef {
        real vector     y
        real matrix     X
        real matrix     Z
        real scalar     vcetype
        string scalar   coding
}
```

In our main program, we define a `struct problemdef` scalar, let's call it `pd`, and then we pass `pd` from one subroutine to the next.

One advantage of this coding style is that, if we later discover that we need to add another element to the structure—say, results are to be optionally projected according to matrix P—splicing in the new feature will be easy. We add the new element to the structure definition

```
struct problemdef {
        real vector     y
        real matrix     X
        real matrix     Z
        real matrix     P           // Projection, optional
        real scalar     vcetype
        string scalar   coding
}
```

and recompile. Then we modify only the subroutines that must use `P`, and `P` will be right at our fingertips when we need it.

Mata's `optimize()` function uses this approach. `optimize()` finds solutions for the minimum or maximum of a function and, in the process, uses an internal structure containing 57 components to track problems. You can see the structure definition by typing `viewsource optimize.mata` at the Stata (not Mata) prompt. In the code the structure has the inelegant name `opt__struct`, with two underscores between `opt` and `struct`, but do not let the name disguise the simplification the structure itself introduces. It is worth taking a look.

Mata's `optimize()` function uses scalar structures. Vectors of structures are often useful in programming data-management tasks. For instance, in a program dealing with disk files, a useful structure might be

```
struct filedesc {
        string scalar    filename
        real scalar      creation_date
        string scalar    filetype
        string scalar    path
}
```

The structure is defined for holding information of one file. In the program, one would use a `struct filedesc` vector to hold information on the collection of files.

3 When not to use structures

If your problem can be solved by writing a Mata function without recourse to subroutines, there is no need for structures. Consider, for instance,

```
struct point {
        real scalar  x, y
}

real scalar distance(x0, y0, x1, y1)
{
        struct point scalar  p0, p1

        p0.x = x0 ; p0.y = y0
        p1.x = x1 ; p1.y = y1
        return(sqrt((p1.x-p0.x)^2 + (p1.y-p0.y)^2))
}
```

The structure added nothing except complication, and this program would be better written as

```
real scalar distance(x0, y0, x1, y1)
{
        return(sqrt((x1-x0)^2 + (y1-y0)^2))
}
```

You might argue that a structure could be beneficial had we used the new `struct point` type in the arguments of `distance()`. Function `distance()` could have been written

```
real scalar distance(struct point p0, struct point p1)
{
        return(sqrt((p1.x-p0.x)^2 + (p1.y-p0.y)^2))
}
```

Here you argue that you have reduced the number of arguments from four to two and improved readability. You have a good argument.

The official StataCorp response is that users of your routines should not be required to use structures because structures require a level of programming knowledge that most users do not have. You have included a hurdle that will prevent some users from using your routine. We at StataCorp use structures but only in hidden ways.

4 Use of structures in hidden ways

In using Mata functions written by StataCorp, you have used structures and never noticed. Other programmers may wish to adopt our style. Obviously, we at StataCorp might write a program that, in its internals, defines a structure, passes the structure to various subroutines, and then returns results that were stored in the structure. That is not what I meant when I said you have used structures and never noticed. There are cases where we have returned a structure to you but did not tell you that it was in fact a structure.

Mata's `optimize()` does this. Say that you want to find the value of (p_1, p_2), which maximizes $y = \exp(-p_1^2 - p_2^2 - p_1p_2 + p_1 - p_2 - 3)$. One way of obtaining the solution is

```
void myfunction(todo, p, y, g, H)
{
        y = exp(-p[1]^2 - p[2]^2 - p[1]*p[2] +
                 p[1] - p[2] - 3)
}

S = optimize_init()
optimize_init_evaluator(S, &myfunction())
optimize_init_which(S, "max")
optimize_init_evaluatortype(S, "d0")
optimize_init_params(S, (0,0))
optimize(S)
```

You can read about `optimize()` in the *Mata Reference Manual* under [M-5] **optimize( )** or by typing `help mata optimize()`. If you try the example, you will find the maximum occurs at $(1, -1)$.

What I want to emphasize here is the role of `S`. In the documentation we will tell you that `S = optimize_init()` is just something you have to do at the start of a problem. We call `S` the problem handle and tell you that, after obtaining `S`, you are to pass `S` to each of the other optimization functions.

We also tell you that if you feel the need to explicitly declare `S`, make it `transmorphic`. We do not tell you that `S` is a structure and that it is the 57-component structure I mentioned earlier. If you list `S`, you will see

```
: S
  0x15f26f0c
```

The value you see may differ, but it certainly does not look like a 57-component structure. Try typing

```
: liststruct(S)
  (output omitted)
```

You will see something different, namely, the 57-element structure.

When we wrote `optimize()`, we did not do anything special to make `S` list itself as 0x15f26f0c; this is just how structures look when you attempt to list them in raw form. The code 0x15f26f0c is the memory address where the structure is stored.

Let's review the calls to the `optimize()` routines, because now knowing that `S` is a structure, we can see how `optimize()` works. To obtain the solution, we suggested you type

```
S = optimize_init()
optimize_init_evaluator(S, &myfunction())
optimize_init_which(S, "max")
optimize_init_evaluatortype(S, "d0")
optimize_init_params(S, (0,0))
optimize(S)
```

(A boldface **S** is used for emphasis.)

When you typed **S** = `optimize_init()`, `optimize_init()` defined a structure, filled it in with default values, and returned it to you. In the subsequent calls, you supplied that same structure as an argument. The various `optimize_init_*(S, ...)` functions modified the information in **S**. Finally, when you typed `optimize(S)`, you unknowingly passed all 57 things `optimize()` needed to define the problem and to perform the requested optimization.

There are three benefits of this design.

1. If we at StataCorp later find that we need to track 58 rather than 57 things, we simply add another element to the structure and make the few modifications

necessary to use the new information. We need not tell you about it, except perhaps to mention a new feature.

2. By hiding from you that `S` is a structure, you cannot declare `S` to be a `struct opt__struct` scalar and so cannot access the information in `S`. More importantly, you cannot accidentally modify any of `S`'s components. To us, the programmers of `optimize()`, `S` contains 57 things. To you, the user, it contains only an odd hexadecimal number such as 0x15f26f0c. If `optimize()` fails to work as advertised, both of us can be certain that it is our fault, not yours.

   We use this same approach at StataCorp to protect us from ourselves. We manage a large code base, and we continually struggle to keep projects separated from one another. Our goal is that Stata internally be a well-defined set of independent modules. We want that so that modifications to one module do not require modifications to others, so we can make improvements to one module and not worry about unanticipated, and usually negative, side effects.

   Consider a project to implement a new estimator that uses `optimize()` as a subroutine. The code we write does not include the declaration `struct opt__struct scalar S`. The new project treats `S` as a transmorphic just as we tell you to do. Thus the new code we write cannot reach into what is properly the purview of `optimize()` and that keeps us from building dependencies where none should exist, which is always a temptation.

   By following this rule, if we need to modify `optimize()`, we can lay our hands on all the relevant code simply by searching for the declaration `struct opt__struct`. We can be certain that code not containing the declaration is irrelevant because the code does not have access to the contents, even if it has access to the variable.

3. This brings us to the third benefit, which is that we can modify our code and that will have no implication for yours. We can make changes in the structure, even large ones, and you do not even have to recompile your code that uses `optimize()`.

5 Mechanics

There are several mechanical issues involved in using structures that I need to tell you about. First and foremost, structures must be defined before they are used. In the program

```
struct point {
        real scalar  x, y
}

real scalar distance(struct point p0, struct point p1)
{
        return(sqrt((p1.x-p0.x)^2 + (p1.y-p0.y)^2))
}
```

you cannot reverse the order of the definitions of `struct point` and `real scalar distance()` because the Mata compiler would not know how to interpret `p1.x`, `p0.x`, etc.

Second, although you can usually omit declarations of variables and arguments, you cannot omit them for structures, or more correctly, you cannot omit them if you need to access what is inside them. It would not do to code the above program

```
real scalar distance(p1, p2)
{
        return(sqrt((p1.x-p0.x)^2 + (p1.y-p0.y)^2))
}
```

because then the Mata compiler would not know that `p0` and `p1` are `struct point`s and thus would not know how to interpret `p1.x`, `p0.x`, etc.

The requirement for explicit declarations also applies to structures used as variables within the program. In the program

```
real scalar distance(x0, y0, x1, y1)
{
        struct point scalar  p0, p1            // <- required

        p0.x = x0 ; p0.y = y0
        p1.x = x1 ; p1.y = y1
        return(sqrt((p1.x-p0.x)^2 + (p1.y-p0.y)^2))
}
```

you cannot omit the `struct point scalar p0, p1`.

Finally, do not omit the `scalar` when it applies to structures. Those with C programming instincts will find this rule difficult to remember. The following will not work:

```
real scalar distance(x0, y0, x1, y1)
{
        struct point         p0, p1          // <- problem here

        p0.x = x0 ; p0.y = y0
        p1.x = x1 ; p1.y = y1
        return(sqrt((p1.x-p0.x)^2 + (p1.y-p0.y)^2))
}
```

The function will compile without error, but an error will be issued when the function is executed. The error will arise because `struct point p0, p1` was interpreted as meaning `struct point matrix p0, p1` rather than as a `struct point scalar`. Why substituting `matrix` for `scalar` causes the problem is going to take some explaining, but it should be obvious to you why Mata assumed `matrix` rather than `scalar`. That was based on Mata's standard rule that has nothing to do with structures: omit the element type, and `transmorphic` is assumed; and omit the storage type, and `matrix` is assumed.

There is another Mata rule, also not specific to structures, that you already know but may never have verbalized:

> The entirety of an object must be defined before a component of the object can be defined.

Applying this rule in standard cases means that you cannot define the element `z[1]` before you have defined the entire vector `z` to be, say, 1×3. Similarly, you cannot define the element `Z[1,1]` before you have defined the entire matrix `Z` to be, for example, 2×3. Declaring `real vector z` or `real matrix Z` is not sufficient. This story is going to end that you cannot define `p0.x` before you define `p0`, but it is going to take a while to get there. Let's explore the standard, nonstructure case first. Everything we learn will apply to the structure case.

There are many ways you define `z` to be 1×3 or `Z` to be 2×3; there are so many ways that you may not even be aware that you are defining the entirety of `z` or `Z` because you do not think of it in that way. For `z`: 1×3, you might simply initialize *all of* `z` to contain the values you want:

```
z = (1, 2, 3)
```

For `Z`: 2×3, you could follow the same approach,

```
Z = (1, 2, 3 \ 4, 5, 6)
```

or perhaps you code something that simply results in `Z`'s entire definition, such as

```
Z = pinv(A)
```

Here you have a preexisting matrix `A`: 3×2, and the entirety of `Z` simply arises from calculation. `pinv()` returns the Moore–Penrose inverse.

The point is that defining the entirety of a vector or matrix object is such a common action you do not usually think of it as something special; it is obvious that you have to define `z` or `Z` before you can redefine `z[1]` or `Z[1,1]`. In some programs, however, `z` and `Z` do not define themselves naturally, and then you use built-in function `J()` to begin. You code

```
z = J(1, 3, 0)

Z = J(2, 3, 0)
```

`J(`r`,` c`,` x`)` returns an $r \times c$ matrix with all elements set to x. Thus `J(1, 3, 0)` returns a 1×3 vector with elements set to 0 and `J(2, 3, 0)` returns a 2×3 matrix with elements similarly set to 0. Then you can proceed to reset the elements,

```
z    = J(1, 3, 0)
z[1] = ...
z[2] = ...
z[3] = ...
```

So what is the effect of including explicit declarations such as `real vector z` and `real matrix Z` in your programs? In the program

```
function example(...)
{
        real vector    z
        real matrix    Z

        ...
}
```

the result is to make `z` be 1×0 and to make `Z` be 0×0 at the outset. The type and general shape are established, but the details have yet to be specified.

Structures work the same way. Let's return to our broken program and understand why it is broken and what we can do about it:

```
real scalar distance(x0, y0, x1, y1)
{
        struct point         p0, p1          // <- problem here

        p0.x = x0 ; p0.y = y0
        p1.x = x1 ; p1.y = y1
        return(sqrt((p1.x-p0.x)^2 + (p1.y-p0.y)^2))
}
```

To remind you, we omitted `scalar` from the `struct point p0, p1` declaration, with the result that `matrix` was assumed. We now know that matrices start their life being 0×0, and matrices of structures are no different from any other kind of matrix. `p0` and `p1` are 0×0. Hence statements such as `p0.x =` ... make no sense, in the same way that `z[1] =` ... makes no sense when `z` is 0×0. There is no first element of `z`, and similarly there is no `x` component of `p0`.

What can we do about it? The easy solution is simply to add the word `scalar` back to the declaration. Then `p0` and `p1` will be 1×1, and our program will work. The other solution would be to set `p0` equal to a 1×1 `struct point`, which is similar to how we solved the previous problems for `z` and `Z`. We called a function, `J()`, that returned a 1×3 real vector and a 2×3 real matrix, respectively. Similarly, we could set `p0` equal to the result from a function that returned a 1×1 `struct point` scalar; that is, a generic solution to our real vector `z` problem was

```
z    = ...
z[1] = ...
z[2] = ...
z[3] = ...
```

In the same way, a generic solution to our structure problem is

```
p0   = ...
p0.x = ...
p0.y = ...
```

The only difference between the two problems is that, in `z =` ..., the dots returned a 1×3 real vector, and in `p0 =` ..., the dots must return a 1×1 `struct point` scalar. I am

about to show you how to do that but, before I do, let me emphasize that no rational person would use this solution because including `scalar` on the original declaration is easier. This solution will have a use, however, when we need a vector or matrix of structures. But let's stay with the 1×1 case at first. The solution is

```
p0   = J(1, 1, point())
p0.x = ...
p0.y = ...
```

The solution is `p0 = J(1, 1, point())`, just as `z = J(1, 3, 0)` was a solution to our real vector problem. The differences are that we do not want a 1×3 result, we want a 1×1 result, so the first two arguments to `J()` are 1 and 1; and we do not want a real result, we want a `struct point` result, so the third argument changes from 0, an instance of a real, to `point()`, an instance of a `struct point`. By the way, the solution simplifies to

```
p0   = point()
p0.x = ...
p0.y = ...
```

because `J(1, 1,` *anything*`)` simplifies to *anything*.

Let me explain about `point()`. When you declare a structure such as

```
struct point {
        real scalar  x, y
}
```

in addition to recording the definition, Mata also creates a function of the same name that returns a 1×1 instance of the structure. Thus when we code `p0 = J(1, 1, point())`, or `p0 = point()`, we obtain a 1×1 `struct point` scalar. Doing that defines the entirety of `p0` and then we can define its components.

As I said, including `scalar` on the declaration line would have been easier. In another problem, however, we might need a 1×3 `struct point` vector, and the solution just given generalizes to that case. We declare `struct point vector p` so that `p` is 1×0, and then we code `p = J(1, 3, point())`, just as we coded `z = J(1, 3, 0)` when we wanted a 1×3 real vector.

For vectors and matrices of structures, there is a variation available to the `J(`r`, `c`, point())` solution. The function `point()`—the function Mata automatically created from our structure definition—allows arguments, and its full syntax is `point(`r`, `c`)`. The r and c are optional. In full form, the function returns an $r \times c$ `struct point` vector or matrix. So rather than coding code `p = J(1, 3, point())`, we can code `p = point(1, 3)`. It does not matter which we code, although the second will execute a little more quickly.

6 Technical notes

What follows is included to reassure you that all the generalities one would expect are included in Mata's implementation of structures. With the exception of items (1)–(4), the issues addressed below seldom arise.

1. Structures and structure references are fully compiled, the latter into *address* + *offset* form. This means structures can be used without concerns about performance degradation. This also means that if a structure definition is modified, all functions that include explicit declarations of the structure must be recompiled.

2. A structure definition `struct` *whatever* `{ ... }` also results in the automatic creation of new function *whatever*`()`. If you are creating `.mlib` libraries, include *whatever*`()` among the functions saved, and similarly, if you are saving your functions in `.mo` files, be sure to save *whatever*`.mo`. Do this even if your other functions do not include calls to *whatever*`()`. The Mata compiler itself will insert calls to *whatever*`()` to construct structure scalars.

3. Let `w` be a `struct` *whatever* scalar and assume that `w` contains real matrix `X`. Then you can use `w.X` just as you can use any other matrix. In particular, `w.X[`i`,`j`]` refers to the i, j-element of matrix `w.X`.

4. Let `w` be a `struct` *whatever* vector and assume that `w` contains real matrix `X`. Then you can use `w[`k`].X` just as you can use any other matrix. In particular, `w[`k`].X[`i`,`j`]` refers to the i, j-element of matrix `w[`k`].X`.

5. Let `w` be a `struct` *whatever* matrix and assume that `w` contains real matrix `X`. Then you can use `w[`k`,`l`].X` just as you can use any other matrix. In particular, `w[`k`,`l`].X[`i`,`j`]` refers to the i, j-element of matrix `w[`k`,`l`].X`.

6. Let `w` be a `struct` *whatever* vector. Then `w[`i`]` is a `struct` *whatever* scalar. `w[`i`]` could be passed as an argument to any function expecting a `struct` *whatever* scalar.

7. Let `w` be a `struct` *whatever* matrix. Then `w[`i`,`j`]` is a `struct` *whatever* scalar. `w[`i`,`j`]` could be passed as an argument to any function expecting a `struct` *whatever* scalar.

   `w[`i`,.]` is a `struct` *whatever* rowvector. `w[`i`,.]` could be passed as an argument to any function expecting a `struct` *whatever* rowvector.

   `w[.,`j`]` is a `struct` *whatever* colvector. `w[.,`j`]` could be passed as an argument to any function expecting a `struct` *whatever* colvector.

8. Structures may contain other structures. For instance,

```
struct point {
        real scalar  x, y
}

struct line {
        struct point scalar p0, p1
}
```

   Let `li` be a `struct line` scalar. Then `li.p0` and `li.p1` are `struct point` scalars. `li.p0` and `li.p1` could be passed as arguments to any function expecting a `struct point` scalar.

`li.p0.x` is a real scalar and presumably is the x coordinate of `p0`.

`li.p0.y` is a real scalar and presumably is the y coordinate of `p0`.

9. Structures may contain structures may contain structures, and so on. `a.b.c.d` refers to the component `d` of structure `c` of structure `b` of structure `a`.

10. Structures may not contain themselves.

11. Structure pointers are allowed but are used less in Mata than in languages such as C. Structure pointers are necessary in Mata when you need structures to contain themselves, just as they are in C. Such constructs are commonly used to construct linked lists.

12. Let `w` be a `struct` *whatever* scalar. Then `&w` is a pointer to the scalar, which is to say, a `pointer(struct` *whatever* `scalar)`. `&(w.x)` is a pointer to the component `x` of the structure.

    Let `w` be a `struct` *whatever* vector. Then `&w` is a pointer to the vector, which is to say, a `pointer(struct` *whatever* `vector)`. `&(w[`i`])` is a `pointer(struct` *whatever* `scalar)`.

    Let `w` be a `struct` *whatever* matrix. Then `&w` is a pointer to the matrix, which is to say, a `pointer(struct` *whatever* `matrix)`. `&(w[`i`,`j`])` is a `pointer(struct` *whatever* `scalar)`.

13. `p->x` refers to component `x` of the structure pointed to by `p` and is equivalent to `(*p).x`. `p` must be explicitly declared a `pointer(struct` *whatever*`) scalar`.

14. Pointers to pointers of structures and pointers to pointers to pointers of structures, etc., are allowed. `(*p)->x` refers to component `x` of the structure pointer pointed to by `p` and is equivalent to `(**p).x`. `p` must be explicitly declared a `pointer(pointer(struct` *whatever*`) scalar) scalar`.

15. Be careful to distinguish between a `pointer(struct` *whatever*`) vector) scalar` and `pointer(struct` *whatever*`) scalar) vector`. `(*p)[i]` is appropriate for the first and `(*(p[i]))`, equivalent to `(*p[i])`, for the second. The same applies to matrices.

16. Memory management, and in particular the allocating and freeing of memory for structures, is automatic. If you do not use pointers to structures, this is simple and easy and works exactly as you would expect. If you do use pointers to structures, this is still simple and easy for you, but you may be surprised at the features provided by Mata. If you use pointers, the memory associated with a structure is not released until (1) the value of the last pointer pointing to the structure is changed or (2) the pointer itself ceases to exist because the program in which the pointer appears returns. For instance, consider a linked list. Let `*p` be a member of the list and assume `p->next` points to the next member. Assume that `p->next` is the only pointer pointing to `*(p->next)`. Simply coding `p->next = NULL` is sufficient to free `*(p->next)`. You may code this even if freeing `*(p->next)`

implies subsequent structures will need to be freed, such as `*(p->next->next)`, `*(p->next->next->next)`, and so on. Mata handles all of this for you.

7 Conclusion

Structures have two important uses: (1) introduction of new types such as complex (if Mata did not already have them) or quaternions (which Mata does not have), and (2) in programming complicated problems, where complicated means lots of information is required simply to describe what is to be done. In both cases, structures make it easy to share information across functions and make code more readable by hiding details. Hidden structures can also be used to enforce separation of complicated systems into independent modules. There is a cost to learning how to use structures, but those who program complicated tools will find the investment profitable.

About the author

William Gould is President of StataCorp, head of development, and principal architect of Mata.