Title

pointers — Pointers

Syntax

```
pointer[ (totype) ] [ orgtype ] [ function ] ...
```

where *totype* is

```
[ eltype ] [ orgtype ] [ function ]
```

and where *eltype* and *orgtype* are

eltype	*orgtype*
`transmorphic`	`matrix`
`numeric`	`vector`
`real`	`rowvector`
`complex`	`colvector`
`string`	`scalar`
`pointer[ (towhat) ]`	

`pointer[ (totype) ] [ orgtype ]` can be used in front of declarations, be they function declarations, argument declarations, or variable definitions.

Description

Pointers are objects that contain the addresses of other objects. The `*` prefix operator obtains the contents of an address. Thus if *p* is a pointer, `*`*p* refers to the contents of the object to which *p* points. Pointers are an advanced programming concept. Most programs, including involved and complicated ones, can be written without them.

In Mata, pointers are commonly used to

1. put a collection of objects under one name and
2. pass functions to functions.

One need not understand everything about pointers merely to pass functions to functions; see [M-2] **ftof**.

Remarks and examples

stata.com

Remarks are presented under the following headings:

What is a pointer?

A pointer is the address of a variable or a function. Say that variable X contains a matrix. Another variable p might contain 137,799,016, and if 137,799,016 were the address at which X were stored, then p would be said to point to X. Addresses are seldom written in base 10, so rather than saying p contains 137,799,016, we would be more likely to say that p contains `0x836a568`, which is the way we write numbers in base 16. Regardless of how we write addresses, however, p contains a number and that number corresponds to the address of another variable.

In our program, if we refer to p, we are referring to p's contents, the number `0x836a568`. The monadic operator `*` is defined as "refer to the contents of the address" or "dereference": `*`p means X. We could code `Y = *p` or `Y = X`, and either way, we would obtain the same result. In our program, we could refer to X`[`i`,`j`]` or `(*`p`)[`i`,`j`]`, and either way, we would obtain the i, j element of X.

The monadic operator `&` is how we put addresses into p. To load p with the address of X, we code p `= &`X.

The special address 0 (zero, written in hexadecimal as `0x0`), also known as `NULL`, is how we record that a pointer variable points to nothing. A pointer variable contains `NULL` or it contains the address of another variable.

Or it contains the address of a function. Say that p contains `0x836a568` and that `0x836a568`, rather than being the address of matrix X, is the address of function f`()`. To get the address of f`()` into p, just as we coded p `= &`X previously, we code p `= &`f`()`. The `()` at the end tells `&` that we want the address of a function. We code `()` on the end regardless of the number of arguments f`()` requires because we are not executing f`()`, we are just obtaining its address.

To execute the function at `0x836a568`—now we will assume that f`()` takes two arguments and call them i and j—we code `(*`p`)(`i`,`j`)` just as we coded `(*`p`)[`i`,`j`]` when p contained the address of matrix X.

Pointers to variables

To create a pointer *p* to a variable, you code

```
p = &varname
```

For instance, if *X* is a matrix,

```
p = &X
```

stores in *p* the address of *X*. Subsequently, referring to `*p` and referring to *X* amount to the same thing. That is, if *X* contained a 3 × 3 identity matrix and you coded

```
*p = Hilbert(4)
```

then after that you would find that *X* contained the 4 × 4 Hilbert matrix. *X* and `*p` are the same matrix.

If *X* contained a 3 × 3 identity matrix and you coded

```
(*p)[2,3] = 4
```

you would then find `X[2,3]` equal to 4.

You cannot, however, point to the interior of objects. That is, you cannot code

```
p = &X[2,3]
```

and get a pointer that is equivalent to `X[2,3]` in the sense that if you later coded `*p=2`, you would see the change reflected in `X[2,3]`. The statement `p = &X[2,3]` is valid, but what it does, we will explain in *Pointers to expressions* below.

By the way, variables can be sustained by being pointed to. Consider the program

```
pointer(real matrix) scalar example(real scalar n)
{
     real matrix   tmp

     tmp = I(3)
     return(&tmp)
}
```

Ordinarily, variable `tmp` would be destroyed when `example()` concluded execution. Here, however, `tmp`'s existence will be sustained because of the pointer to it. We might code

```
p = example(3)
```

and the result will be to create `*p` containing the 3 × 3 identity matrix. The memory consumed by that matrix will be freed when it is no longer being pointed to, which will occur when *p* itself is freed, or, before that, when the value of *p* is changed, perhaps by

```
p = NULL
```

For a discussion of pointers to structures, see [M-2] **struct**. For a discussion of pointers to classes, see [M-2] **class**.

Pointers to expressions

You can code

```
p = &(2+3)
```

and the result will be to create *p containing 5. Mata creates a temporary variable to contain the evaluation of the expression and sets p to the address of the temporary variable. That temporary variable will be freed when p is freed or, before that, when the value of p is changed, just as tmp was freed in the example in the previous section.

When you code

```
p = &X[2,3]
```

the result is the same. The expression is evaluated and the result of the expression stored in a temporary variable. That is why subsequently coding *p=2 does not change X[2,3]. All *p=2 does is change the value of the temporary variable.

Setting pointers equal to the value of expressions can be useful. In the following code fragment, we create n 5×5 matrices for later use:

```
pvec = J(1, n, NULL)
for (i=1; i<=n; i++) pvec[i] = &(J(5, 5, .))
```

Pointers to functions

When you code

```
p = &functionname()
```

the address of the function is stored in p. You can later execute the function by coding

```
... (*p)(...)
```

Distinguish carefully between

```
p = &functionname()
```

and

```
p = &(functionname())
```

The latter would execute *functionname*() with no arguments and then assign the returned result to a temporary variable.

For instance, assume that you wish to write a function neat() that will calculate the derivative of another function, which function you will pass to neat(). Your function, we will pretend, returns a real scalar.

You could do that as follows

```
real scalar neat(pointer(function) p, other args...)
{
     ...
}
```

although you could be more explicit as to the characteristics of the function you are passed:

```
real scalar neat(pointer(real scalar function) p, other args...)
{
        ...
}
```

In any case, inside the body of your function, where you want to call the passed function, you code

```
(*p)(arguments)
```

For instance, you might code

```
approx = ( (*p)(x+delta)-(*p)(x) ) / delta
```

The caller of your `neat()` function, wanting to use it with, say, function `zeta_i_just_wrote()`, would code

```
result = neat(&zeta_i_just_wrote(), other args...)
```

For an example of the use of pointers in sophisticated numerical calculations, see Støvring (2007).

Pointers to pointers

Pointers to pointers (to pointers ...) are allowed, for instance, if *X* is a matrix,

```
p1 = &X
p2 = &p1
```

Here `*p2` is equivalent to *p1*, and `**p2` is equivalent to *X*.

Similarly, we can construct a pointer to a pointer to a function:

```
q1 = &f()
q2 = &p1
```

Here `*q2` is equivalent to *q1*, and `**q2` is equivalent to *f*`()`.

When constructing pointers to pointers, never type `&&`—such as `&&`*x*—to obtain the address of the address of *x*. Type `&(&`*x*`)` or `& &`*x*. `&&` is a synonym for `&`, included for those used to coding in C.

Pointer arrays

You may create an array of pointers, such as

```
P = (&X1, &X2, &X3)
```

or

```
Q = (&f1(), &f2(), &f3())
```

Here `*P[2]` is equivalent to *X2* and `*Q[2]` is equivalent to *f2*`()`.

Mixed pointer arrays

You may create mixed pointer arrays, such as

```
R = (&X, &f())
```

Here `*R[2]` is equivalent to `f()`.

You may not, however, create arrays of pointers mixed with real, complex, or string elements. Mata will abort with a type-mismatch error.

Definition of NULL

`NULL` is the special pointer value that means "points to nothing" or undefined. `NULL` is like 0 in many ways—for instance, coding `if (X)` is equivalent to coding `if (X!=NULL)`, but `NULL` is distinct from zero in other ways. 0 is a numeric value; `NULL` is a pointer value.

Use of parentheses

Use parentheses to make your meaning clear.

In the table below, we assume

```
p = &X
P = (&X11, &X12 \ &X21, &X22)
q = &f()
Q = (&f11(), &f12() \ &f21(), &f22())
```

where *X*, *X11*, *X12*, *X21*, and *X22* are matrices and `f()`, `f11()`, `f12()`, `f21()`, and `f22()` are functions.

Expression	Meaning
`*p`	*X*
`*p[1,1]`	*X*
`(*p)[1,1]`	*X*`[1,1]`
`*P[1,2]`	*X12*
`(*P[1,2])[3,4]`	*X12*`[3,4]`
`*q(a,b)`	execute function `q()` of `a`, `b`; dereference that
`(*q)(a,b)`	*f*(*a*,*b*)
`(*q[1,1])(a,b)`	*f*(*a*,*b*)
`*Q[1,2](a,b)`	nonsense
`(*Q[1,2])(a,b)`	*f12*(*a*,*b*)

Pointer arithmetic

Arithmetic with pointers (which is to say, with addresses) is not allowed:

```
: y = 2
: x = &y
: x+2
                    <stmt>:  3012  undefined operation on pointer (e.g., p1>p2)
r(3012);
```

Do not confuse the expression `x+2` with the expression `*x+2`, which is allowed and in fact evaluates to 4.

You may use the equality and inequality operators `==` and `!=` with pairs of pointer values:

```
if (p1 == p2) {
      ...
}

if (p1 != p2) {
      ...
}
```

Also pointer values may be assigned and compared with the value `NULL`, which is much like, but still different from, zero: `NULL` is a 1×1 scalar containing an address value of 0. An unassigned pointer has the value `NULL`, and you may assign the value `NULL` to pointers:

```
p = NULL
```

Pointer values may be compared with `NULL`,

```
if (p1 == NULL) {
      ...
}

if (p1 != NULL) {
      ...
}
```

but if you attempt to dereference a `NULL` pointer, you will get an error:

```
: x = NULL
: *x + 2
                    <stmt>:  3010  attempt to dereference NULL pointer
r(3010);
```

Concerning logical expressions, you may directly examine pointer values:

```
if (p1) {
      ...
}
```

The above is interpreted as if `if (p1!=NULL)` were coded.

Listing pointers

You may list pointers:

```
: y = 2
: x = &y
: x
  0x8359e80
```

What is shown, `0x8359e80`, is the memory address of `y` during our Stata session. If you typed the above lines, the address you would see could differ, but that does not matter.

Listing the value of pointers often helps in debugging because, by comparing addresses, you can determine where pointers are pointing and whether some are pointing to the same thing.

In listings, `NULL` is presented as `0x0`.

Declaration of pointers

Declaration of pointers, as with all declarations (see [M-2] **declarations**), is optional. That basic syntax is

`pointer`[`(`*totype*`)`] *orgtype* [`function`] ...

For instance,

`pointer(real matrix) scalar` *p1*

declares that *p1* is a pointer scalar and that it points to a real matrix, and

`pointer(complex colvector) rowvector` *p2*

declares that *p2* is a rowvector of pointers and that each pointer points to a complex colvector, and

`pointer(real scalar function) scalar` *p3*

declares that *p3* is a pointer scalar, and that it points to a function that returns a real scalar, and

`pointer(pointer(real matrix function) rowvector) colvector` *p4*

declares that *p4* is a column vector of pointers to pointers, the pointers to which each element points are rowvectors, and each of those elements points to a function returning a real matrix.

You can omit the pieces you wish.

`pointer() scalar` *p5*

declares that *p5* is a pointer scalar—to what being uncertain.

`pointer scalar` *p5*

means the same thing.

`pointer` *p6*

declares that *p6* is a pointer, but whether it is a matrix, vector, or scalar, is unsaid.

Use of pointers to collect objects

Assume that you wish to write a function in two parts: `result_setup()` and `repeated_result()`.

In the first part, `result_setup()`, you will be passed a matrix and a function by the user, and you will make a private calculation that you will use later, each time `repeated_result()` is called. When `repeated_result()` is called, you will need to know the matrix, the function, and the value of the private calculation that you previously made.

One solution is to adopt the following design. You request the user code

```
resultinfo = result_setup(setup args...)
```

on the first call, and

```
value = repeated_result(resultinfo, other args...)
```

on subsequent calls. The design is that you will pass the information between the two functions in `resultinfo`. Here `resultinfo` will need to contain three things: the original matrix, the original function, and the private result you calculated. The user, however, never need know the details, and you will simply request that the user declare `resultinfo` as a pointer vector.

Filling in the details, you code

```
pointer vector result_setup(real matrix X, pointer(function) f)
{
     real matrix     privmat
     pointer vector  info

     ...
     privmat = ...
     ...
     info = (&X, f, &privmat)
     return(info)
}

real matrix repeated_result(pointer vector info, ...)
{
     pointer(function) scalar   f
     pointer(matrix)   scalar   X
     pointer(matrix)   scalar   privmat

     f = info[2]
     X = info[1]
     privmat = info[3]

     ...
     ... (*f)(...) ...
     ... (*X) ...
     ... (*privmat) ...
     ...
}
```

It was not necessary to unload `info[]` into the individual scalars. The lines using the passed values could just as well have read

```
... (*info[2])(...) ...
... (*info[1]) ...
... (*info[3]) ...
```

Efficiency

When calling subroutines, it is better to pass the evaluation of pointer scalar arguments rather than the pointer scalar itself, because then the subroutine can run a little faster. Say that `p` points to a real matrix. It is better to code

```
... mysub(*p) ...
```

rather than

```
... mysub(p) ...
```

and then to write `mysub()` as

```
function mysub(real matrix X)
{
    ... X ...
}
```

rather than

```
function mysub(pointer(real matrix) scalar p)
{
    ... (*p) ...
}
```

Dereferencing a pointer (obtaining $*p$ from p) does not take long, but it does take time. Passing $*p$ rather than p can be important if `mysub()` loops and performs the evaluation of $*p$ hundreds of thousands or millions of times.

Diagnostics

The prefix operator `*` (called the dereferencing operator) aborts with error if it is applied to a nonpointer object.

Arithmetic may not be performed on undereferenced pointers. Arithmetic operators abort with error.

The prefix operator `&` aborts with error if it is applied to a built-in function.

References

Gould, W. W. 2011. Advanced Mata: Pointers. The Stata Blog: Not Elsewhere Classified. http://blog.stata.com/2011/09/11/advanced-mata-pointers/.

Støvring, H. 2007. A generic function evaluator implemented in Mata. *Stata Journal* 7: 542–555.

Also see

[M-2] **intro** — Language definition