Title

stata.com

findexternal( ) — Find, create, and remove external globals

Description

`findexternal(`*name*`)` returns a pointer (see [M-2] **pointers**) to the external global matrix, vector, or scalar whose name is specified by *name*, or to the external global function if the contents of *name* end in `()`. `findexternal()` returns `NULL` if the external global is not found.

`crexternal(`*name*`)` creates a new external global 0×0 real matrix with the specified name and returns a pointer to it; it returns `NULL` if an external global of that name already exists.

`rmexternal(`*name*`)` removes (deletes) the specified external global or does nothing if no such external global exists.

`nameexternal(`*p*`)` returns the name of `*`*p*.

Syntax

```
pointer() scalar  findexternal(string scalar name)

pointer() scalar  crexternal(string scalar name)

void              rmexternal(string scalar name)

string    scalar  nameexternal(pointer() scalar p)
```

Remarks and examples

stata.com

Remarks are presented under the following headings:

Also see *Linking to external globals* in [M-2] **declarations**.

Definition of a global

When you use Mata interactively, any variables you create are known, equivalently, as externals, globals, or external globals.

```
: myvar = x
```

Such variables can be used by subsequent functions that you run, and there are two ways that can happen:

```
function example1(...)
{
        external real myvar

        ... myvar ...
}
```

and

```
function example2(...)
{
        pointer(real) p

        p = findexternal("myvar")
        ... *p ...
}
```

Using the first method, you must know the name of the global at the time you write the source code, and when you run your program, if the global does not exist, it will refuse to run (abort with `myvar` not found). With the second method, the name of the global can be specified at run time and what is to happen when the global is not found is up to you.

In the second example, although we declared `p` as a pointer to a real, `myvar` will not be required to contain a real. After `p = findexternal("myvar")`, if `p!=NULL`, `p` will point to whatever `myvar` contains, whether it be real, complex, string, or another pointer. (You can diagnose the contents of `*p` using `eltype(*p)` and `orgtype(*p)`; see [M-5] **eltype( )**.)

Use of globals

Globals are useful when a function must remember something from one call to the next:

```
function example3(real scalar x)
{
        pointer() scalar p

        if ( (p = findexternal("myprivatevar")) == NULL) {
                printf("you haven't called me previously")
                p = crexternal("myprivatevar")
        }
        else {
                printf("last time, you said "%g", *p)
        }
        *p = x
}
```

```
: example3(2)
you haven't called me previously
: example3(31)
last time, you said 2
: example3(9)
last time, you said 31
```

Note our use of the name myprivatevar. It actually is not a private variable; it is global, and you would see the variable listed if you described the contents of Mata's memory. Because global variables are so exposed, it is best that you give them long and unlikely names.

In general, programs do not need global variables. The exception is when a program must remember something from one invocation to the next, and especially if that something must be remembered from one invocation of Mata to the next.

When you do need globals, you probably will have more than one thing you will need to recall. There are two ways to proceed. One way is simply to create separate global variables for each thing you need to remember. The other way is to create one global pointer vector and store everything in that. In the following example, we remember one scalar and one matrix:

```
function example4()
{
        pointer(pointer() vector) scalar   p
        scalar                             s
        real matrix                        X
        pointer() scalar                   ps, pX

        if ( (p = findexternal("mycollection")) == NULL) {
                ... calculate scalar s and X from nothing ...
                ... and save them:
                p = crexternal("mycollection")
                *p = (&s, &X)
        }
        else {
                ps = (*p)[1]
                pX = (*p)[2]
                ... calculate using *ps and *pX ...
        }
}
```

In the above example, even though crexternal() created a 0×0 real global, we morphed it into a 1×2 pointer vector:

```
p = crexternal("mycollection")      *p is 0 × 0 real
*p = (&s, &X)                       *p is 1 × 2 vector
```

just as we could with any nonpointer object.

In the else part of our program, where we use the previous values, we do not use variables s and X, but ps and pX. Actually, we did not really need them, we could just as well have used *((*p)[1]) and *((*p)[2]), but the code is more easily understood by introducing *ps and *pX.

Actually, we could have used the variables s and X by changing the else part of our program to read

```
else {
        s = *(*p)[1]
        X = *(*p)[2]
        ... calculate using s and X ...
        *p = (&s, &X)            ← remember to put them back
}
```

Doing that is inefficient because s and X contain copies of the global values. Obviously, the amount of inefficiency depends on the sizes of the elements being copied. For s, there is really no inefficiency at all because s is just a scalar. For X, the amount of inefficiency depends on the dimensions of X. Making a copy of a small X matrix would introduce just a little inefficiency.

The best balance between efficiency and readability is achieved by introducing a subroutine:

```
function example5()
{
        pointer(pointer() vector) scalar   p
        scalar                             s
        real matrix                        X

        if ( (p = findexternal("mycollection")) == NULL) {
                example5_sub(1, s=., X=J(0,0,.))
                p = crexternal("mycollection")
                *p = (&s, &X)
        }
        else {
                example5_sub(0, (*p)[1], (*p)[2])
        }
}

function example5_sub(scalar firstcall, scalar x, matrix X)
{
        ...
}
```

The last two lines in the not-found case

```
p = crexternal("mycollection")
*p = (&s, &X)
```

could also be coded

```
*crexternal("mycollection") = (&s, &X)
```

Conformability

`findexternal(`*name*`)`, `crexternal(`*name*`)`:
name: 1×1
result: 1×1

`rmexternal(`*name*`)`:
name: 1×1
result: *void*

`nameexternal(`*p*`)`:
p: 1×1
result: 1×1

Diagnostics

`findexternal(`*name*`)`, `crexternal(`*name*`)`, and `rmexternal(`*name*`)` abort with error if *name* contains an invalid name.

`findexternal(`*name*`)` returns `NULL` if *name* does not exist.

`crexternal(`*name*`)` returns `NULL` if *name* already exists.

`nameexternal(`*p*`)` returns `""` if p = `NULL`. Also, `nameexternal()` may be used not just with pointers to globals but pointers to locals as well. For example, you can code `nameexternal(&myx)`, where `myx` is declared in the same program or a calling program. `nameexternal()` will usually return the expected local name, such as "`myx`". In such cases, however, it is also possible that `""` will be returned. That can occur because, in the compilation/optimization process, the identity of local variables can be lost.

Also see

[M-5] **valofexternal( )** — Obtain value of external global

[M-4] **programming** — Programming functions