The Stata Journal

The *Stata Journal* publishes reviewed papers together with shorter notes or comments, regular columns, book reviews, and other material of interest to Stata users. Examples of the types of papers include 1) expository papers that link the use of Stata commands or programs to associated principles, such as those that will serve as tutorials for users first encountering a new field of statistics or a major new technique; 2) papers that go "beyond the Stata manual" in explaining key features or uses of Stata that are of interest to intermediate or advanced users of Stata; 3) papers that discuss new commands or Stata programs of interest either to a wide spectrum of users (e.g., in data management or graphics) or to some large segment of Stata users (e.g., in survey statistics, survival analysis, panel analysis, or limited dependent variable modeling); 4) papers analyzing the statistical properties of new or existing estimators and tests in Stata; 5) papers that could be of interest or usefulness to researchers, especially in fields that are of practical importance but are not often included in texts or other journals, such as the use of Stata in managing datasets, especially large datasets, with advice from hard-won experience; and 6) papers of interest to those who teach, including Stata with topics such as extended examples of techniques and interpretation of results, simulations of statistical concepts, and overviews of subject areas.

For more information on the *Stata Journal*, including information for authors, see the web page

http://www.stata-journal.com

The *Stata Journal* is indexed and abstracted in the following:

- CompuMath Citation Index®
- RePEc: Research Papers in Economics
- Science Citation Index Expanded (also known as SciSearch®)

Users of any of the software, ideas, data, or other materials published in the *Stata Journal* or the supporting files understand that such use is made without warranty of any kind, by either the *Stata Journal*, the author, or StataCorp. In particular, there is no warranty of fitness of purpose or merchantability, nor for special, incidental, or consequential damages such as loss of profits. The purpose of the *Stata Journal* is to promote free communication among Stata users.

The *Stata Journal*, electronic version (ISSN 1536-8734) is a publication of Stata Press. Stata and Mata are registered trademarks of StataCorp LP.

The Stata Journal (2009)
9, Number 4, pp. 599–620

Mata Matters: File processing

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. The subject of this column is using Mata to read into Stata datasets that are formatted difficultly, which involves Mata's file processing and string processing capabilities.

Keywords: pr0049, Mata, I/O, string processing, file processing, structures

1 Introduction

We wish to read the information contained in the following file into Stata:

begin wd47839.txt

```
AutoWeatherScan Report                                     page 1

Report for College Station generated 10/19/2009

   10/19/2009 09:00:00
                    Humidity:    69%
                    Dew Point:   49F
                    Temperature: 60.2F
                    Wind:        2.3 mph from the WSW
                    Wind Gust:   7.8 mph
              17:00:00
                    Humidity:    66%
                    Dew Point:   49F
                    Temperature: 62.2F
                    Wind:        2.5 mph from the WSW
                    Wind Gust:   5.0 mph
   10/18/2009 09:00:00
                    Humidity:    69%
                    Dew Point:   49F
                    Temperature: 60.3F
^L
AutoWeatherScan Report                                     page 2
                    Wind:        2.3 mph from the WSW
                    Wind Gust:   7.8 mph

^L
```

end wd47839.txt

File wd47839.txt contains fictional weather reports on measurements from College Station, Texas at three different times, 9:00 and 17:00 on 10/19 and 9:00 on 10/18. I ask you to imagine that the above file is an extract of a much larger file. Regardless of size, the file was intended to be printed rather than infiled into other software. If the file

pr0049

were printed, each `^L` (*control L*) would turn into a page break. Control characters are just one of the problems we will face in getting these data into Stata.

Assume that we wish to extract temperature and wind but not wind gust or the other information. We wish file `wd47839.txt` contained

```
"College Station" "10/19/2009 09:00:00" 60.2 "WSW" 2.3
"College Station" "10/19/2009 17:00:00" 62.2 "WSW" 2.5
"College Station" "10/18/2009 09:00:00" 60.3 "WSW" 2.3
```

because then all we would have to do is type

```
. infile  str20 location  str20 dtime  temp  str10 wind_dir  wind_speed
> using desired.raw
(3 observations read)

. gen double datetime = clock(dtime, "MDYhms")

. format datetime %tc

. drop dtime

. list

     +---------------------------------------------------------------------+
     |        location   temp   wind_dir   wind_s~d              datetime |
     |---------------------------------------------------------------------|
  1. | College Station   60.2        WSW        2.3   19oct2009 09:00:00 |
  2. | College Station   62.2        WSW        2.5   19oct2009 17:00:00 |
  3. | College Station   60.3        WSW        2.3   18oct2009 09:00:00 |
     +---------------------------------------------------------------------+
```

To make the problem even more difficult, assume that we have many files (imagine hundreds) all formatted just like `wd47839.txt`. For the purposes of the example, we have two additional files, file `wd47840.txt` and file `wd82332.txt`:

begin `wd47840.txt`

```
AutoWeatherScan Report                                      page 1

Report for College Station generated 10/18/2009

   10/18/2009 12:00:00
                    Humidity:    80%
                    Dew Point:   63F
                    Temperature: 69.3F
                    Wind:        3.4 mph from the West
                    Wind Gust:   9.8 mph
```

end `wd47840.txt`

begin `wd82332.txt`

```
AutoWeatherScan Report                                      page 1

Report for Boston generated 10/19/2009

   10/19/2009 14:00:00
                    Humidity:    50%
                    Dew Point:   36F
                    Temperature: 60.2F
                    Wind:        7 mph from the NE
                    Wind Gust:   0.0 mph
^L
```

end `wd82332.txt`

Our problem is to write code to read these files and produce a new file—let's call it `desired.raw`—containing

begin `desired.raw`

```
"College Station" "10/19/2009 09:00:00" 60.2 "WSW" 2.3
"College Station" "10/19/2009 17:00:00" 62.2 "WSW" 2.5
"College Station" "10/18/2009 09:00:00" 60.3 "WSW" 2.3
"College Station" "10/18/2009 12:00:00" 69.3 "West" 3.4
"Boston" "10/19/2009 14:00:00" 60.2 "NE" 7
```

end `desired.raw`

As a preview, once we have written the code, we will produce the desired result by typing

```
. mata:  driver("wd*.txt", "desired.raw")
```

2 Outline of solution

Mata is excellent at processing files and processing strings, and I will show you how to do that. Before getting into the programming details, however, let's outline the solution.

1. The top-level routine—we will call it `driver()`—will open output file `desired.raw`. `driver()` will then find all the `wd*.txt` files and, one at a time, feed each to our second-level routine, `process_file()`.
2. Second-level routine `process_file()` will open the input file and read the lines in it. Each line will be sent to the third-level routine, `process_line()`.
3. Third-level routine `process_line()` will look at the line to determine if it is of interest. If the line is of interest, `process_line()` will interpret (parse) it and collect the desired information. Once `process_line()` has collected sufficient information to form an output line, `process_line()` will output it.

Routines 1 and 2 will be easy to write. Here is the `driver()` routine:

```
void driver(string scalar filespec, string scalar output_filename)
{
        string colvector        filenames
        real scalar             i
        real scalar             output_fh

        filenames = dir(".", "files", filespec)

        output_fh = fopen(output_filename, "w")
        for (i=1; i<=length(filenames); i++) {
                process_file(filenames[i], output_fh)
        }
        fclose(output_fh)
}
```

Throughout this column, I am going to use strict Mata syntax, although the above routine could just as well be coded

```
void driver(filespec, output_filename)
{
        filenames = dir(".", "files", filespec)

        output_fh = fopen(output_filename, "w")
        for (i=1; i<=length(filenames); i++) {
                process_file(filenames[i], output_fh)
        }
        fclose(output_fh)
}
```

The routine may look less forbidding using Mata's looser syntax, but including the declarations makes the routine easier to understand, makes it so that Mata can compile more efficient code, and makes it less likely that I will make an error.

We have already seen `driver()` in action; as mentioned, we will invoke our system by typing

```
. mata:  driver("wd*.txt", "desired.raw")
```

Thus input argument `filespec` will be `"wd*.txt"` and input argument `output_filename` will be `"desired.raw"`.

`driver()` begins by placing in string vector `filenames` the names of the `wd*.txt` files; see `help mata dir()`. Next `driver()` opens new file `desired.raw` for output and records its file handle in `output_fh`; see `help mata fopen()`. Then `driver()` loops across the filenames and calls `process_file()` with each when it executes the code

```
        for (i=1; i<=length(filenames); i++) {
                process_file(filenames[i], output_fh)
        }
```

The line

```
        for (i=1; i<=length(filenames); i++) {
```

specifies the looping. It says to begin by setting `i` equal to 1. The last part, `i++`, says that `i` is to be incremented by 1 at the end of each loop. The `for` statement could just as well have been written

```
        for (i=1; i<=length(filenames); i=i+1) {
```

except that Mata executes `i++` faster than `i=i+1`.

The middle part, `i <= length(filenames)`, says that the loop is to be executed so long as `i <= length(filenames)` is true. In our example, `filenames` will be the string vector `("wd47839.txt", "wd47840.txt", "wd82332.txt")`, and thus its `length()` is 3. Thus `process_file(filenames[i], output_fh)` will be executed three times, first with `i=1`, then with `i=2`, and finally with `i=3`, meaning that `process_file()` will be called with `"wd47839.txt"`, then with `"wd47840.txt"`, and finally with `"wd82332.txt"`.

After the loop, `process_file()` ends with

```
        fclose(output_fh)
```

fclose() is a standard Mata function; see help mata fclose(). To read and write files in Mata, first you fopen() them, then you use fget() or fread() to read from them—or use fput() or fwrite() to write into them—and finally you fclose() them. fopen() returns a file handle, really just an integer, and you use that file handle (integer) as an argument in the other I/O commands so that they know which file they should read from, write into, or close.

3 Construction

I will create file code.do containing the Mata code. At this point in the development, code.do looks like this:

begin code.do

```
cscript
set matastrict on
mata:
void driver(string scalar filespec, string scalar output_filename)
{
        string colvector        filenames
        real scalar             i
        real scalar             output_fh
        filenames = dir(".", "files", filespec)
        output_fh = fopen(output_filename, "w")
        for (i=1; i<=length(filenames); i++) {
                process_file(filenames[i], output_fh)
        }
        fclose(output_fh)
}
end
```

end code.do

Note the top line, cscript. cscript (Gould 2001) is a hidden jewel in Stata that I often use instead of clear all. cscript is one of Stata's undocumented commands. Undocumented is not a wholly precise term because undocumented commands are not really undocumented; you can even type help cscript to see its documentation! cscript is one of the commands we at StataCorp use in certifying Stata, and I find it useful when I am developing code, too. You can learn about the other undocumented commands by typing help undocumented. Anyway, think of cscript as clear all on steroids. If cscript resets too much for your tastes, substitute clear all.

I said that I would use Mata's strict syntax, and that is not just intent on my part; I will ask Stata to enforce my intent. set matastrict on puts Mata in strict mode. In the default nonstrict mode, Mata would not complain if I entered

(*Continued on next page*)

```
void driver(filespec, output_filename)
{
        filenames = dir(".", "files", filespec)
        output_fh = fopen(output_filename, "w")
        for (i=1; i<=length(filenames); i++) {
                process_file(filenames[i], output_fh)
        }
        fclose(output_fh)
}
```

With `matastrict` on, Mata would respond with three error messages:

```
variable filenames undeclared
variable output_fh undeclared
variable i undeclared
(0 lines skipped)
r(3000);
```

I find those messages useful because when I am explicitly declaring my variables, any such complaint usually indicates an error in my thinking or my typing; either way, it is a bug and would be difficult to track down later.

4 Solution

Next we are going to write `process_file()`. The verbal description from our outline of the solution is "Second-level routine `process_file()` will open the input file and read the lines in it. Each line will be sent to the third-level routine, `process_line()`." I am about to show you `process_file()` in its final form; it is a straightforward routine except that you will see one line in it that may make you scream, "What is that? `struct`!?? I was hoping never to learn about that!" If that is your response, then ignore the line for the time being. It was not even in the first draft of the routine, but that `struct` is going to simplify our solution. Here is the final version of `process_file()`:

```
void process_file(string scalar filename, real scalar output_fh)
{
        struct myproblem scalar         pr
        real scalar                     input_fh
        initialize_record(pr.wr)
        pr.output_fh = output_fh
        input_fh = fopen(filename, "r")
        while ( (pr.line=fget(input_fh)) != J(0,0,"") ) {
                process_line(pr)
        }
        output_record(pr)
        fclose(input_fh)
}
```

The original draft of `process_file()`, however, looked like this:

```
void process_file(string scalar filename, real scalar output_fh)
{
        real scalar                     input_fh
        string scalar                   line

        input_fh = fopen(filename, "r")
        while ( (line=fget(input_fh)) != J(0,0,"") ) {
                process_line(line, output_fh, ...)
        }
        fclose(input_fh)
}
```

Note the ellipsis (...) in the call to `process_line()`. That ellipsis actually appeared in the draft, so the above code will not compile. My drafts often look like this.

The part that is written, however, deals with opening the input file, reading the lines, and passing them one at a time to `process_line()`. We have opened files before. This time, the line to open the file reads

```
        input_fh = fopen(filename, "r")
```

whereas when we wrote `driver()`, the line read

```
        output_fh = fopen(output_filename, "w")
```

This time, we are opening a file for input rather than output, so the second argument changes from `"w"` to `"r"`, which stand for write and read, respectively. See `help mata fopen()`.

The loop in our code to read all the lines and call `process_line()` with each reads

```
        while ( (line=fget(input_fh)) != J(0,0,"") ) {
                process_line(line, output_fh, ...)
        }
```

The `while` statement specifies that the loop is to continue as long as

```
                (line=fget(input_fh)) != J(0,0,"")
```

`(line=fget(input_fh)) != J(0,0,"")` is called a compound expression, and it can be confusing the first time you see it, so let me take an aside and explain.

5 An aside on compound expressions

Did you know that you could code

```
a = b = c
```

For instance, you can code

```
a = b = 0
```

to set both a and b to zero. In the same way, the compound expression a = b = c means to set both a and b to c. If you code a = b = c, Mata interprets it as

```
a = (b = c)
```

Mata is willing to see an assignment anywhere in an expression. We usually think of assignments being of the form, for instance,

```
a = b + 1
```

If we also needed to set b = c + d, we can code

```
b = c + d
a = b + 1
```

or we can code

```
a = (b = c + d) + 1
```

In the same way, assignments can appear inside any expression, such as

```
(a = b) != c
```

The above expression assigns the value in b to a and then compares that with c. The expression returns 1 if the result is not equal to c.

6 Development continues

We were discussing the expression

```
(line=fget(input_fh)) != J(0,0,"")
```

which appeared in

```
while ( (line=fget(input_fh)) != J(0,0,"") ) {
```

That expression is identical in form to (a = b) != c. (line=fget(input_fh)) != J(0,0,"") assigns the result from fget(input_fh) to line, and then it compares that with J(0,0,""). The result from the expression is either true (1) or false (0).

Thus the loop in process_file(),

```
while ( (line=fget(input_fh)) != J(0,0,"") ) {
        process_line(line, output_fh, ...)
}
```

obtains input by using fget(input_fh), stores it in line, and continues to do that as long as fget(input_fh) does not return J(0,0,""). It is a property of fget() that it returns J(0,0,"") (a 0 × 0 string matrix) when there are no more lines in the file; see help mata fget().

With the draft of process_file() in hand, I drafted process_line(), and it looked something like this:

```
void process_line(...)
{
        if (process_line_reportfor(...)) return
        if (process_line_datetime(...)) return
        if (process_line_time(...)) return
        if (process_line_temperature(...)) return
        if (process_line_wind(...)) return
        /* otherwise, we ignore the line */
}
```

The underlying idea of this routine is that it would call separate subroutines that would look at the line and determine if they could act on it. The first one that could act on it would cause `process_line()` to return. If none of the subroutines could process the line, the line would be ignored. For instance, `process_line_reportfor()` would look to see if the line looked like

```
Report for College Station generated 10/19/2009
```

If the line looks like that, somehow I would hold on to the "College Station" part and `process_line_reportfor()` would return a 1, meaning "I have processed this line." Routine `process_line()` would see the 1 and would return, because the calling code is

```
        if (process_line_reportfor(...)) return
```

If the line did not match "Report for", `process_line_reportfor()` would do nothing and return 0, and thus `process_line()` would execute its next line of code,

```
if (process_line_datetime(...)) return
```

`process_line_datetime()` would ask whether the line looked like

```
10/19/2009 09:00:00
```

If it did, `process_line_datetime()` would hold on to the date and time so that they could be output later, and `process_line_datetime()` would return 1. If the line did not match, `process_line_datetime()` would return 0, and that would cause `process_line()` to call the next routine.

The next routine would be `process_line_time()`, and so the process would continue until `process_line()` ran out of routines, at which point the input line would be ignored.

Having convinced myself that my approach would work and that I more-or-less knew how to code it, it was time to think about the arguments and data flow. The `process_line_*()` routines would need to hold on to information if they matched the line. That information would be output later once the information was complete. So where to store the information? The answer was obvious to me, and from now on, it will be obvious to you: store the information in a structure. Structures are single-name objects that can hold many different things within them. Here is the definition of the structure that I would need for holding on to the weather information:

```
struct weather_record {
        real scalar     has_data            /* Boolean */
        string scalar   station_name
        string scalar   date
        string scalar   time
        real scalar     temperature
        string scalar   wind_direction
        real scalar     wind_speed
}
```

Remember that an output record is supposed to look like this:

```
"College Station" "10/19/2009 09:00:00" 60.2 "WSW" 2.3
```

Thus I would need to hold on to the station_name (College Station), the date (10/19/2009), the time (09:00:00), the temperature (60.2), the wind_direction (WSW), and the wind_speed (2.3). Also, I added one more variable, has_data, because I knew I would need it later when I got to coding the actual output of the records. I was thinking ahead, but ignore has_data if you wish. Even delete it. A wonderful feature of structures is that you can go back and add more variables to them and leave the rest of your code unchanged!

Having defined what a weather_record is, I can create weather_record variables. In a program, I could create a weather_record variable called george by declaring

```
        struct weather_record scalar    george
```

Because george is a variable, and because variables can be passed to subroutines, I will be able to call subroutines with george and thus pass all the information recorded in george to them. Any routine that has access to george has access to all of its elements, which are referred to as george.has_data, george.station_name, george.date, and so on.

So I modified my code to include the new structure, although the result was still more in the form of notes than code that would compile:

begin code.do

```
cscript

set matastrict on
mata:

void driver(string scalar filespec, string scalar output_filename)
{
        string colvector        filenames
        real scalar             i
        real scalar             output_fh

        filenames = dir(".", "files", filespec)

        output_fh = fopen(output_filename, "w")
        for (i=1; i<=length(filenames); i++) {
                process_file(filenames[i], output_fh)
        }
        fclose(output_fh)
}
```

```
struct weather_record {
        real scalar     has_data             /* Boolean */
        string scalar   station_name
        string scalar   date
        string scalar   time
        real scalar     temperature
        string scalar   wind_direction
        real scalar     wind_speed
}

void process_file(string scalar filename, real scalar output_fh)
{
        real scalar                     input_fh
        string scalar                   line
        struct weather_record scalar    wr

        input_fh = fopen(filename, "r")
        while ( (line=fget(input_fh)) != J(0,0,"") ) {
                process_line(line, output_fh, wr)
        }
        fclose(input_fh)
}

void process_line(..., struct weather_record scalar wr)
{
        if (process_line_reportfor(..., wr)) return
        if (process_line_datetime(..., wr)) return
        if (process_line_time(..., wr)) return
        if (process_line_temperature(..., wr)) return
        if (process_line_wind(..., wr)) return
        /* otherwise, we ignore the line */
}

real scalar process_line_reportfor(..., struct weather_record scalar wr)
{
        ...
}

real scalar process_line_datetime(..., struct weather_record scalar wr)
{
        ...
}

...
end
```

end code.do

The ellipses in the above are real; they appeared in the draft. I knew I would have to pass more information to routines such as process_line_reportfor() and process_line_datetime() than just wr. Variable wr was just where the weather record was being stored. At a minimum, the routines would need to see the line read from the file.

What other variables, I wondered, would I need to pass to them? Rather than figure that out, I decided to create another structure containing all the information that would be necessary. I mentioned how wonderful structures are because you can go back and add another variable to them without having to modify your code, except to make use of the additional information. By adding a structure, if I found I omitted something, I could add it later. This new structure I defined as

```
struct myproblem {
        struct weather_record scalar  wr
        string scalar                 line
        real scalar                   output_fh
}
```

Note that this structure contains my `weather_record` structure! Structure `myproblem` would contain everything in one variable that I would need to pass from `process_line()` to the `process_line_*()` subroutines. My code now looked like this:

begin `code.do`

```
cscript

set matastrict on
mata:

void driver(string scalar filespec, string scalar output_filename)
{
        string colvector        filenames
        real scalar             i
        real scalar             output_fh

        filenames = dir(".", "files", filespec)

        output_fh = fopen(output_filename, "w")
        for (i=1; i<=length(filenames); i++) {
                process_file(filenames[i], output_fh)
        }
        fclose(output_fh)
}

struct weather_record {
        real scalar     has_data             /* Boolean */
        string scalar   station_name
        string scalar   date
        string scalar   time
        real scalar     temperature
        string scalar   wind_direction
        real scalar     wind_speed
}

struct myproblem {
        struct weather_record scalar  wr
        string scalar                 line
        real scalar                   output_fh
}

void process_file(string scalar filename, real scalar output_fh)
{
        struct myproblem scalar         pr
        real scalar                     input_fh

        initialize_record(pr.wr)
        pr.output_fh = output_fh

        input_fh = fopen(filename, "r")
        while ( (pr.line=fget(input_fh)) != J(0,0,"") ) {
                process_line(pr)
        }
        output_record(pr)
        fclose(input_fh)
}
```

```
void process_line(struct myproblem scalar pr)
{
        if (process_line_reportfor(pr)) return
        if (process_line_datetime(pr)) return
        if (process_line_time(pr)) return
        if (process_line_temperature(pr)) return
        if (process_line_wind(pr)) return
        /* otherwise, we ignore the line */
}

real scalar process_line_reportfor(struct myproblem scalar pr)
{
        ...
}

real scalar process_line_datetime(struct myproblem scalar pr)
{
        ...
}

...
```

end code.do

There is one thing in the above I want to call to your attention. Below I have marked the line I want to emphasize:

```
void process_file(string scalar filename, real scalar output_fh)
{
        struct myproblem scalar         pr
        real scalar                     input_fh

        initialize_record(pr.wr)
        pr.output_fh = output_fh        // <-- NEW

        input_fh = fopen(filename, "r")
        while ( (pr.line=fget(input_fh)) != J(0,0,"") ) {
                process_line(pr)
        }
        output_record(pr)
        fclose(input_fh)
}
```

In the new code, all the information passed by process_file() to process_line() is contained in struct myproblem scalar pr. Among that information is pr.output_fh. Routine process_file(), however, received output_fh as an argument, outside the structure. We must copy the value output_fh to pr.output_fh, and that is the line I added and nearly forgot. This would be easier to understand had I called pr.output_fh something different, such as george. Then the new line would read

```
        pr.george    = output_fh        // <-- NEW
```

Different names would make it clear to you that input argument output_fh and structure member pr.george are different things, and if I want to have the value recorded in output_fh also recorded in pr.george, then I obviously must copy it there. The same applies whether the member is named pr.george or pr.output_fh. Naming the member the same as the argument is actually better style because the identical names emphasizes the relationship, even though the identical names can be confusing the first time you see them.

Let me show one of the `process_line_*()` routines in full detail. They are not particularly interesting. Remember, a `process_line_*()` routine is supposed to identify whether the line is of interest and, if it is, record the relevant information in our `weather_record` structure and return 1, meaning that the line has been interpreted. Otherwise, the routine does nothing and returns 0. Here is `process_line_reportfor()`:

```
real scalar process_line_reportfor(struct myproblem scalar pr)
{
        string scalar    work
        real scalar      loc
        /*
            ----+----1----+----
            Report for ___________ generated ...
        */
        if (substr(pr.line, 1, 11)!="Report for ") return(0)
        work = substr(pr.line, 12, .)
        if ((loc = strpos(work, "generated"))==0) return(0)
        pr.wr.station_name = strtrim(substr(work, 1, loc-1))
        return(1)
}
```

In the code, we are looking to see if the line is of the form "Report for ___________ generated ...". If the line is of that form, we extract the ___________, store that in `pr.wr.station_name`, and return 1. Otherwise, we do nothing and return 0.

There are two `process_line_*()` routines that are of special interest, however. We have yet to address how and when the record stored in `pr.wr` will be output to file `desired.raw` (file handle `pr.output_fh`). Determining when to output is particularly difficult in this weather data example and is typically a problem when processing files intended for human eyes rather than for programmer convenience. Looking at the output, we do not know it is time to output a record until the next one starts. In the weather data, we want to output a record for each date/time, and when we do that is when we see another date/time. There are two such lines that prompt us that it is time to output a record: when we see a new date-and-time record,

```
10/19/2009 09:00:00
```

and when we see a new time-only record,

```
           09:00:00
```

Thus I included output logic in the code for `process_line_datetime()` and in the code for `process_line_time()`. Here is the code for processing a date-and-time record:

```
real scalar process_line_datetime(struct myproblem scalar pr)
{
        string rowvector    piece

        piece = tokens(pr.line)
        if (length(piece)!=2) return(0)
        if (!looks_like_date(piece[1])) return(0)
        if (!looks_like_time(piece[2])) return(0)

        output_record(pr)
        reinitialize_record(pr.wr)
        pr.wr.has_data = 1
        pr.wr.date     = piece[1]
        pr.wr.time     = piece[2]
        return(1)
}
```

The above is the full code, but what I want you to see is

```
real scalar process_line_datetime(struct myproblem scalar pr)
{
        Determine if not a date/time record
        return 0 if it is not

        output_record(pr)
        reinitialize_record(pr.wr)

        pr.wr.has_data = 1
        pr.wr.date     = piece[1]
        pr.wr.time     = piece[2]
        return(1)
}
```

We output the existing record *before* filling it in with new values, and before filling it in, we clear the record's previous values.

The above code would be more readable if, rather than containing the single line

```
        output_record(pr)
```

it contained

```
        if (we have data in our record) {
                output_record(pr)
        }
```

Instead, I put the if-we-have-data logic in `output_record()` itself. That way of coding is safer. `output_record()` reads

```
void output_record(struct myproblem scalar pr)
{
        if (pr.wr.has_data == 0) return

        fput(pr.output_fh, sprintf(`""%s" "%s %s" %g "%s" %g"',
                        pr.wr.station_name,
                        pr.wr.date, pr.wr.time,
                        pr.wr.temperature,
                        pr.wr.wind_direction, pr.wr.wind_speed))
}
```

We know when we have data to output if `pr.wr.has_data` is not 0. In this design, it is the responsibility of any routine that *starts* a record to set `pr.wr.has_data` to 1. I decided that I would not worry whether a record was complete; it would be sufficient if the record was started. I argued that completeness of data is more the responsibility of data analysis, more the responsibility of the researcher using Stata than it is of a processing routine designed to deliver an accurate rendition of the underlying data.

The other `process_line_*()` routine that required modification for outputting of records was `process_line_time()`, which reads

```
real scalar process_line_time(struct myproblem scalar pr)
{
        string rowvector    piece
        string scalar       hold

        piece = tokens(pr.line)
        if (length(piece)!=1) return(0)
        if (!looks_like_time(piece)) return(0)

        output_record(pr)

        hold = pr.wr.date
        reinitialize_record(pr.wr)
        pr.wr.has_data = 1
        pr.wr.date     = hold
        pr.wr.time     = piece
        return(1)
}
```

Just as with `process_line_datetime()`, the above is the full code, but what I want you to see is

```
real scalar process_line_time(struct myproblem scalar pr)
{
        Determine if not a date/time record
        return 0 if it is not

        hold = pr.wr.date                <- hold on to date
        reinitialize_record(pr.wr)
        pr.wr.has_data = 1
        pr.wr.date     = hold            <- put date back
        pr.wr.time     = piece
        return(1)
}
```

The code here is nearly identical with that of `process_line_datetime()` except that we must carry over the date from the prior record, which means that we must hold on to the date before calling `reinitialize_record()`, and then put the date back afterward. By the way, the code for `reinitialize_record()` reads

```
void reinitialize_record(struct weather_record scalar wr)
{
        wr.has_data = 0
        wr.date = wr.time = ""
        wr.temperature = .
        wr.wind_direction = ""
        wr.wind_speed = .
}
```

And except for some minor details, that is all there is to it. The entire code reads

begin code.do

```
cscript

set matastrict on
mata:

void driver(string scalar filespec, string scalar output_filename)
{
        string colvector        filenames
        real scalar             i
        real scalar             output_fh

        filenames = sort(dir(".", "files", filespec),1)

        output_fh = fopen(output_filename, "w")
        for (i=1; i<=length(filenames); i++) {
                process_file(filenames[i], output_fh)
        }
        fclose(output_fh)
}

struct weather_record {
        real scalar     has_data             /* Boolean */
        string scalar   station_name
        string scalar   date
        string scalar   time
        real scalar     temperature
        string scalar   wind_direction
        real scalar     wind_speed
}

struct myproblem {
        struct weather_record scalar  wr
        string scalar                 line
        real scalar                   output_fh
}

void initialize_record(struct weather_record scalar wr)
{
        wr.station_name = ""
        reinitialize_record(wr)
}

void reinitialize_record(struct weather_record scalar wr)
{
        wr.has_data = 0
        wr.date = wr.time = ""
        wr.temperature = .
        wr.wind_direction = ""
        wr.wind_speed = .
}

void output_record(struct myproblem scalar pr)
{
        if (pr.wr.has_data == 0) return

        fput(pr.output_fh, sprintf(`""%s" "%s %s" %g "%s" %g"',
                        pr.wr.station_name,
                        pr.wr.date, pr.wr.time,
                        pr.wr.temperature,
                        pr.wr.wind_direction, pr.wr.wind_speed))
}
```

```
void process_file(string scalar filename, real scalar output_fh)
{
        struct myproblem scalar         pr
        real scalar                     input_fh

        initialize_record(pr.wr)
        pr.output_fh = output_fh

        input_fh = fopen(filename, "r")
        while ( (pr.line=fget(input_fh)) != J(0,0,"") ) {
                process_line(pr)
        }
        output_record(pr)
        fclose(input_fh)
}

void process_line(struct myproblem scalar pr)
{
        if (process_line_reportfor(pr)) return
        if (process_line_datetime(pr)) return
        if (process_line_time(pr)) return
        if (process_line_temperature(pr)) return
        if (process_line_wind(pr)) return
        /* otherwise, we ignore the line */
}

real scalar process_line_reportfor(struct myproblem scalar pr)
{
        string scalar    work
        real scalar      loc

        /*
            ----+----1----+----
            Report for ___________ generated ...
        */

        if (substr(pr.line, 1, 11)!="Report for ") return(0)
        work = substr(pr.line, 12, .)
        if ((loc = strpos(work, "generated"))==0) return(0)

        pr.wr.station_name = strtrim(substr(work, 1, loc-1))
        return(1)
}

real scalar process_line_datetime(struct myproblem scalar pr)
{
        string rowvector    piece

        piece = tokens(pr.line)
        if (length(piece)!=2) return(0)
        if (!looks_like_date(piece[1])) return(0)
        if (!looks_like_time(piece[2])) return(0)

        output_record(pr)
        reinitialize_record(pr.wr)
        pr.wr.has_data = 1
        pr.wr.date     = piece[1]
        pr.wr.time     = piece[2]
        return(1)
}

real scalar process_line_time(struct myproblem scalar pr)
{
        string rowvector    piece
        string scalar       hold
        piece = tokens(pr.line)
        if (length(piece)!=1) return(0)
        if (!looks_like_time(piece)) return(0)

        output_record(pr)

        hold = pr.wr.date
        reinitialize_record(pr.wr)
        pr.wr.has_data = 1
        pr.wr.date     = hold
        pr.wr.time     = piece
        return(1)
}

real scalar process_line_temperature(struct myproblem scalar pr)
{
        string rowvector    piece
        string scalar       s
        real scalar         temp

        /*
           Temperature:  ##.#F
        */

        piece = tokens(pr.line)
        if (length(piece)!=2) return(0)
        if (piece[1]!="Temperature:") return(0)

        s = substr(piece[2], 1, strlen(piece[2])-1)
        pr.wr.temperature = strtoreal(s)
        return(1)
}

real scalar process_line_wind(struct myproblem scalar pr)
{
        string rowvector    piece

        /*
           Wind:  [##]#.# mph from the DIR
        */

        piece = tokens(pr.line)
        if (length(piece)!=6) return(0)
        if (piece[1]!="Wind:") return(0)
        if (piece[3] != "mph") return(0)
        if (piece[4] != "from") return(0)
        if (piece[5] != "the") return(0)

        pr.wr.wind_speed     = strtoreal(piece[2])
        pr.wr.wind_direction = piece[6]
        return(1)
}

real scalar looks_like_date(string scalar original)
{
        string scalar      s
        real scalar        i

        /* #[#]/#[#]/#### */

        s = strtrim(original)

        i = strpos(s, "/")
        if (i<=1 | i>3) return(0)
        if (!isnumeric(substr(s, 1, i-1))) return(0)

        s = substr(s, i+1, .)
        i = strpos(s, "/")
        if (i<=1 | i>3) return(0)
        if (!isnumeric(substr(s, 1, i-1))) return(0)

        s = substr(s, i+1, .)
        if (strlen(s)!=4) return(0)
        if (!isnumeric(s)) return(0)
        return(1)
}

real scalar looks_like_time(string scalar original)
{
        string scalar      s
        real scalar        i

        /* #[#]:##:## */

        s = strtrim(original)

        i = strpos(s, ":")
        if (i<=1 | i>3) return(0)
        if (!isnumeric(substr(s, 1, i-1))) return(0)

        s = substr(s, i+1, .)         /* ##:## */
        if (substr(s, 3, 1)!=":") return(0)
        if (!isnumeric(substr(s, 1, 2))) return(0)
        if (!isnumeric(substr(s, 4, 2))) return(0)
        return(1)
}

real scalar isnumeric(string scalar s)
{
        real scalar        i, len
        string scalar      c

        len = strlen(s)
        if (len==0) return(0)
        for (i=1; i<=len; i++) {
                c = substr(s, i, 1)
                if (c<"0" | c>"9") return(0)
        }
        return(1)
}

end
```

end code.do

7 Using code.do

To run this code on the wd*.dta datasets, we type

```
. do code
  (output omitted)

. mata:  driver("wd*.txt", "desired.raw")

. _
```

The result of running it with the three sample files is

```
. type desired.raw
"College Station" "10/19/2009 09:00:00" 60.2 "WSW" 2.3
"College Station" "10/19/2009 17:00:00" 62.2 "WSW" 2.5
"College Station" "10/18/2009 09:00:00" 60.3 "WSW" 2.3
"College Station" "10/18/2009 12:00:00" 69.3 "West" 3.4
"Boston" "10/19/2009 14:00:00" 60.2 "NE" 7
```

I also ran the code on one hundred `wd*.txt` datasets, each 367,032 bytes, amounting to 734,000 printed pages, and it ran in 21 seconds, meaning the code processed 34,952 pages per second.

This solution required 219 lines of Mata code, counting blank lines, but not counting blank lines between routines. Here is a breakdown

```
function driver()               14
struct   myproblem               5
struct   weather_record          9
function process_file()         15
function process_line()          9
function process_line_*()       89

function initialize_record()     5
function reinitialize_record()   8
function output_record()        10

function looks_like_date()      23
function looks_like_time()      19
function isnumeric()            13
------------------------------------
                               219
```

I did not discuss the last three routines in the text. They were used as subroutines by the `process_line_*()` routines and are included in the listing. These three routines are general utilities that may be of interest to some readers.

The approach we used to process the files is general across most formats of output, even output that looks very different from the weather data example. If you wanted to modify these routines to process different file formats, you would need to modify

```
struct   weather_record
function process_line()
function process_line_*()

function initialize_record()
function reinitialize_record()
function output_record()
```

It would not be difficult. The other routines would remain unchanged.

8 Conclusion

From a programming perspective, I hope this article emphasized the following:

1. Mata has significant file processing capabilities. Learn about `dir()`, `fopen()`, `fget()`, `fput()`, and `fclose()`. Mata has many other file capabilities—see `help m4 io`—but those five functions are sufficient to handle most problems.

2. Mata has significant string processing capabilities; see `help m4 string`.

3. Structures are a programming tool worth learning. They group together related variables, reduce the number of arguments required by subroutines, make code more readable, and make code more modifiable. See `help m2 struct`.

It was not previously mentioned, but I hope you notice that the routines written in solving this problem were short; there were merely a lot of them. That style is recommended. Mata has virtually no overhead for subroutine calls. Short, well-defined subroutines are easier to write, easier to debug, and easier to maintain.

9 Reference

Gould, W. 2001. Statistical software certification. *Stata Journal* 1: 29–50.

About the author

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