L12: Debugging, Error Handling

Also: cell and struct arrays

Lucas A. J. Bastien

E7 Spring 2017, University of California at Berkeley

February 13, 2017

Version: release

Announcements

Lab 04 is due on February 17 at 12 pm (noon)

Reading for this week:

- Chapter 2 (section 2.4 and 2.5), Chapter 8, Chapter 9

Today:

- Debugging
- Error handling
- More data structures: `struct` arrays and `cell` arrays

Wednesday:

- Binary representation of data

Friday:

- Discussion

Make debugging easier

To reduce the number of bugs and to make debugging easier:

- Plan your code ahead on paper
  - What algorithms?
  - What data structures?
- Use a modular approach: divide your code into functions, where each function performs a specific task. Test each function thoroughly
- Write code that is easy to understand and revise
  - Include comments to explain your code (don't over-comment!)
  - Use self-explanatory names for variables
  - Define variables instead of using magic numbers
- Test your code frequently as you write it (every two or three lines)
  - Don't write 30 lines of code without testing anything!

Avoid using magic numbers

Magic number: a numerical value that is used inside of the code, without being defined in a variable

A piece of code that uses magic numbers:

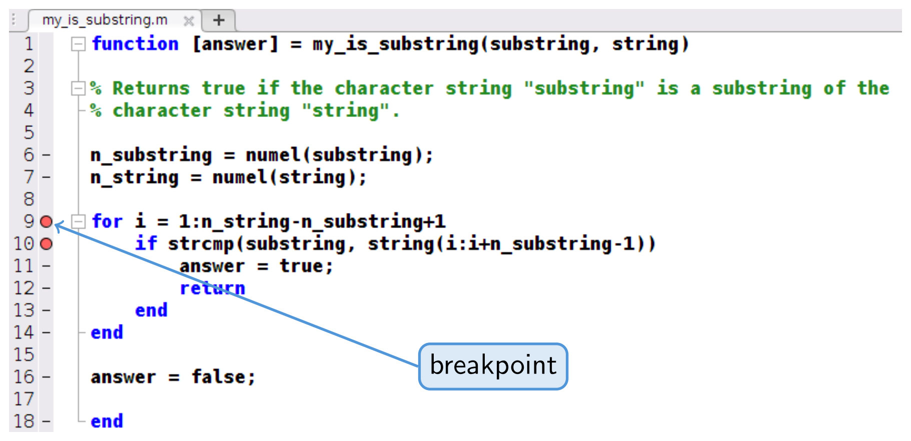
```
>> a = rand(35, 50);
>> total = 0;
>> for i = 1:35
>>     for j = 1:50
>>         total = total + a(i,j);
>>     end
>> end
```

A version that does not use magic numbers:

```
>> n_rows = 35;
>> n_cols = 50;
>> a = rand(n_rows, n_cols);
>> total = 0;
>> for i = 1:n_rows
>>     for j = 1:n_cols
>>         total = total + a(i,j);
>>     end
>> end
```

Using Matlab's debugger

1. Set breakpoints by clicking on the dashes near the line numbers. Breakpoints show as red dots



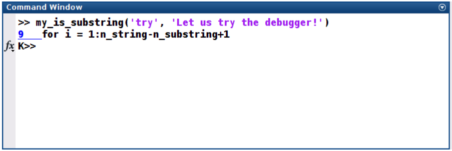
```
function [answer] = my_is_substring(substring, string)

% Returns true if the character string "substring" is a substring of the
% character string "string".

n_substring = numel(substring);
n_string = numel(string);

for i = 1:n_string-n_substring+1
    if strcmp(substring, string(i:i+n_substring-1))
        answer = true;
        return
    end
end

answer = false;

end
```

breakpoint

Alternatively, use the `keyboard` command

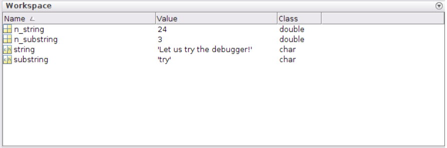
Using Matlab's debugger (continued)

2. Call the function as normal. Matlab will stop at the next breakpoint (or `keyboard` command) and will give you a command prompt labeled as "K>>". From there, **you can type commands as usual, except that you have access to the function's workspace**, as opposed to the regular workspace



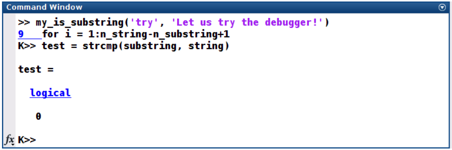
Command Window

```
>> my_is_substring('try', 'Let us try the debugger!')
9   for i = 1:n_string-n_substring+1
K>>
```



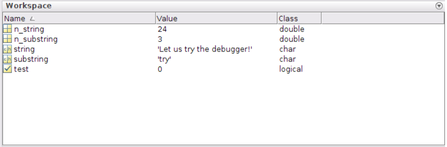
Workspace

Name	Value	Class
n_string	24	double
n_substring	3	double
string	'Let us try the debugger!'	char
substring	'try'	char



Command Window

```
>> my_is_substring('try', 'Let us try the debugger!')
9   for i = 1:n_string-n_substring+1
K>> test = strcmp(substring, string)

test =

  logical

   0

K>>
```

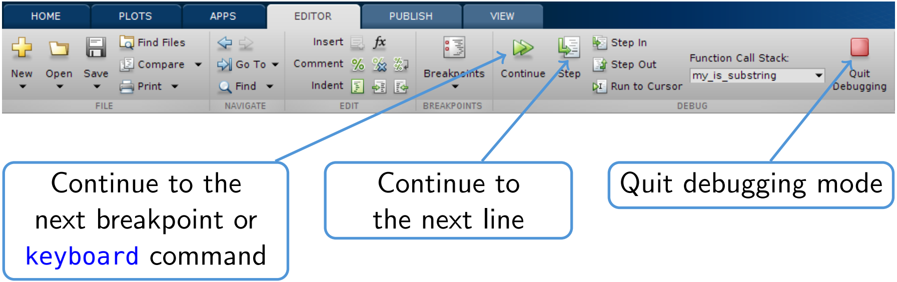


Workspace

Name	Value	Class
n_string	24	double
n_substring	3	double
string	'Let us try the debugger!'	char
substring	'try'	char
test	0	logical

Using Matlab's debugger (continued)

3. From the "EDITOR" menu:
   - Continue to the next breakpoint or `keyboard` command; or
   - Continue to the next line; or
   - Quit the debugging mode



Continue to the next breakpoint or `keyboard` command

Continue to the next line

Quit debugging mode

Try/catch statements

Try/catch statements are used to have code handle errors “gracefully”, as opposed to have Matlab stop the execution of the code. The syntax is:

```
>> try
>>     % Here goes some code that might
>>     % generate an error. The execution
>>     % jumps to the "catch"  block as soon
>>     % as an error occurs
>> catch e
>>     % Here goes some code that will be executed
>>     % if an error occurs in the "try" block. If no
>>     % error occurs in the "try" block, this part of
>>     % the code will not be executed. The variable
>>     % named "e" (you can choose another name) will
>>     % contain information about the error that occurred:
>>     % e.message: the error message (a character string)
>>     % e.identifier: the error identifier (a character string)
>> end
```

Note: specifying a variable to store information about the error is optional

Try/catch statements: example

```
function [result] = my_multiply(a, b)

% Returns the matrix multiplication of a and b if possible,
% and the element-wise multiplication of a and b otherwise.
% If none of these multiplications is possible, this
% function throws an error.

try
    result = a * b;
catch
    try
        result = a .* b;
    catch
        error('None of these multiplications is possible.')
    end
end

end
```

Note: you can manually throw an error using the `error` command

Try/catch statements: practice question

Assuming we start with an empty workspace, what will the value of the variable “v” be after executing the following code?

```
>> array = [2, 5, 0, 1];
>> try
>>     v = array(10);
>> catch e
>>     if strcmp(e.message, 'Matrix dimensions must agree.')
>>         v = Inf;
>>     else
>>         v = NaN;
>>     end
>> end
```

(A) `0`

(B) `Inf`

(C) [✓ `NaN`]

(D) The variable “v” will not be defined

Cell arrays

- All elements in an array of class `double` are of class `double`
- All elements in an array of class `logical` are of class `logical`
- All elements in an array of class `char` are of class `char`

A cell array is a "special" type of array where each element can be of a different class (`double`, `char`, `logical`, `function_handle`, `cell`, `struct`, etc.)

Use curly braces `{}` (instead of square brackets `[]`) to create cell arrays. Both curly braces `{}` and parentheses `()` can be used to index cell arrays (they yield different results)

See the diary for how to create and use cell arrays

Struct arrays

In a `struct` array, each cell contains “fields”. The field names are the same for all cells of the `struct` array. The values of the fields can differ from one cell to the next, and can be of any class (`double`, `char`, `logical`, `function_handle`, `cell`, `struct`, etc.). Valid field names are valid variable names. See the diary for `struct` array syntax

	column 1

row 1	`name:` `'ENGIN 7'`
	`units:` `4`
	`lecture_location:` `'Dwinelle 155'`

row 2	`name:` `'MATH 1A'`
	`units:` `4`
	`lecture_location:` `'VLSB 2050'`

row 3	`name:` `'MATH 1B'`
	`units:` `4`
	`lecture_location:` `'Dwinelle 155'`

Practice question

Assuming that we start with an empty workspace, what will the classes of the variables “v1”, “v2”, and “v3” be, respectively, after executing the following code?

```
>> b(1).value1 = {@cos, @sin, {@tan}};
>> b(1).value2 = 10;
>> a = {[10, 45; -1, -2]; b};
>> v1 = a(1);
>> v2 = a{2};
>> v3 = b(1).value1{1};
```

(A) `double`, `struct`, `function_handle`

(B) `cell`, `cell`, `cell`

(C) `double`, `struct`, `function_handle`

(D) [✓ `cell`, `struct`, `function_handle`]