MATLAB ENGINE C++ GUIDE

Setting the MATLAB engine

First of all, delete the example source code file (engwindemo.cpp) and create your own file called like you want and start write your stuff.

a. Including "engine.h" is necessary to use the Engine.

b. Declare a type pointer to Engine variable and initialize it with

```
Engine *engOpen(const char *startcmd)
```

(the function return a pointer to an engine handle, or NULL if the open fails)
Checking the status of the engine pointer before going on is a good practise and useful to avoid crashes.

Note: read the syntax on the documentation to understand the cause of NULL argument in following code.

c. You must close the engine at the end of your code using

```
int engClose(Engine *ep)
```

(0 stands for success, 1 otherwise)

IMPORTANT TIP: Check the return of Engine functions if you want to avoid hours of brainstress!!!

Writing few lines you start a MATLAB engine session that you can handle as you prefer.

Interact with the MATLAB engine

It's time to ask Matlab something to do.

KEY FUNCTION:

```
int engEvalString(Engine *ep, const char *string)
```

Through this you can invokes commands in Matlab Engine kinda you're using it in interpreter mode.

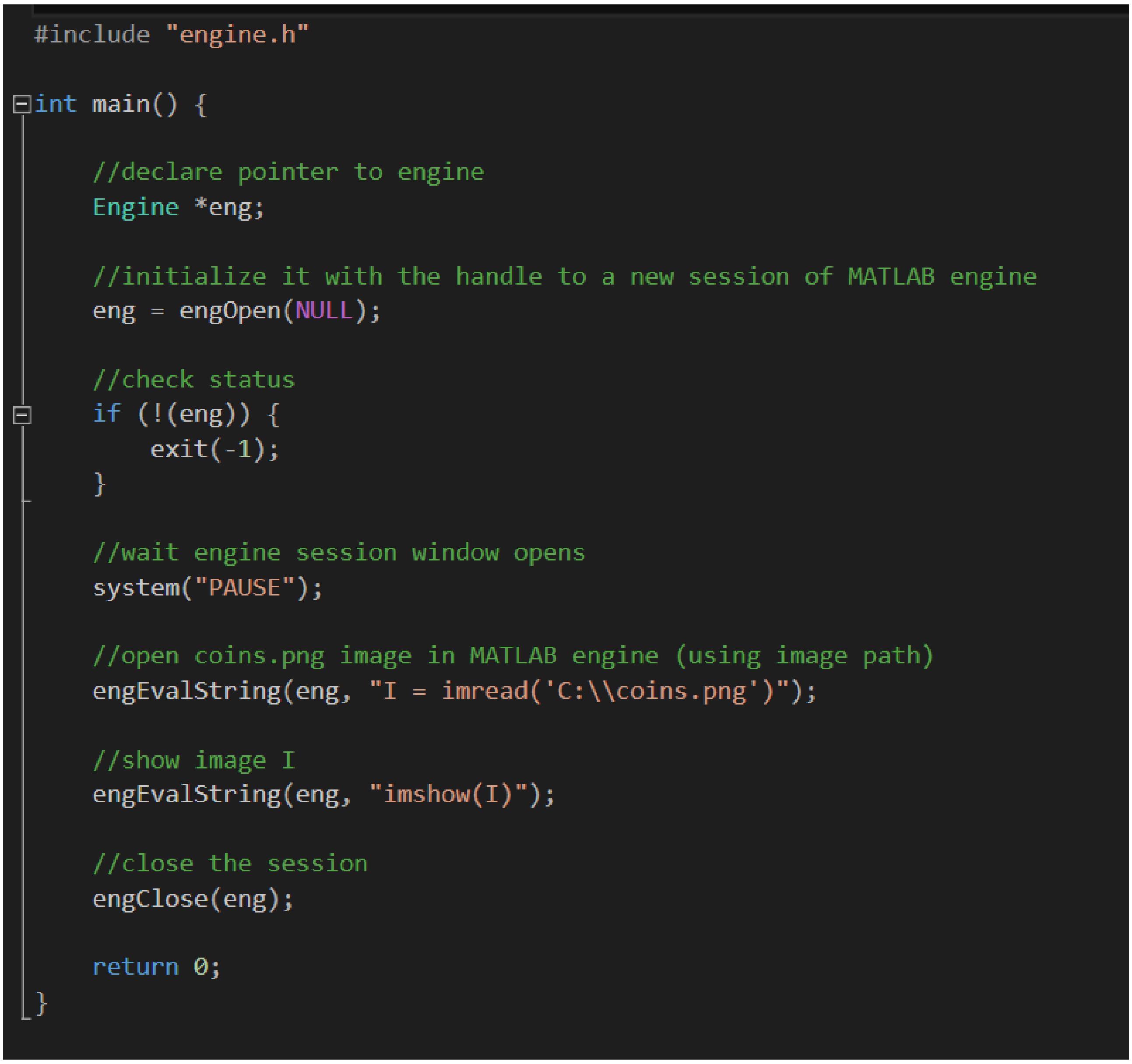
The function takes engine pointer and command string as arguments and returns 1 if the engine session is no longer running or the engine pointer is invalid or NULL.
Otherwise, returns 0 even if the MATLAB engine session cannot evaluate the command.

(You can understand that checking opening status of the engine and the return of this function are good logical tricks to avoid problems during execution)

Note: wait the Engine opens before sending it commands maybe using workaround like system("pause")

EXAMPLE

Now let's see a simple code which opens an engine session and tell MATLAB to read an image and show it.



```cpp
#include "engine.h"

int main() {

    //declare pointer to engine
    Engine *eng;

    //initialize it with the handle to a new session of MATLAB engine
    eng = engOpen(NULL);

    //check status
    if (!(eng)) {
        exit(-1);
    }

    //wait engine session window opens
    system("PAUSE");

    //open coins.png image in MATLAB engine (using image path)
    engEvalString(eng, "I = imread('C:\\coins.png')");

    //show image I
    engEvalString(eng, "imshow(I)");

    //close the session
    engClose(eng);

    return 0;
}
```

lab_example1.cpp

Now it’s easy to send commands to MATLAB but probably it’s not enough for your purpose. Sending commands in a “blackbox” without be able to manage what you’re doing in your workspace is not so useful.

The Engine has many methods that give you the tools to operate on your data in the workspace with an exceptional level of detail.

SOME USEFUL CONCEPTS

1. MXARRAY

It is the fundamental type for MATLAB data.

The mxArray structure contains the following information about the array:

- Its type
- Its dimensions
- The data associated with this array
- If numeric, whether the variable is real or complex
- If sparse, its indices and nonzero maximum elements
- If a structure or object, the number of fields and field names

You can access all info of the array/matrix and its elements using functions ad hoc (that takes a pointer to mxArray as argument).

You can initialize it with functions called `mxCreate*` like (mxCreateNumericArray , mxCreateDouble, etc..), usually use as arguments dimensions of the array to be created and datatype.

It's a good practise to destroy mxArray after used it, `mxDestroyArray` can do this.

2. PUT AND GET

You can create an mxArray and you can populate it coherently but then?

Probably you want to send it to the MATLAB engine to create a variable on the workspace of your session to operate on it with MATLAB tools.

KEY FUNCTION:

```
int engPutVariable(Engine *ep, const char *name, const mxArray
      *pm);
```

(0 if successful and 1 if an error occurs)

As you can imagine through this function you can PUT your mxArray into the workspace. If the mxArray does not exist in the workspace, the function creates it. If an mxArray with the same name exists in the workspace, the function replaces the existing mxArray with the new mxArray.

As you put a mxArray into the workspace, you can get one from it. You need

KEY FUNCTION:

```
mxArray *engGetVariable(Engine *ep, const char *name);
```

(Pointer to a newly allocated mxArray structure, or NULL if the attempt fails)

Note: You can get a mxArray put into the workspace using engPutVariable but you can also extract easily a mxArray created through command sent with engEvalString.

The point is that the array you handle in your MATLAB workspace into classical software usage are mxArray if you manage them in code. You can let MATLAB engine create them by itself invoking commands or you can create them and send into the workspace.

3. ACCESS MXARRAY DATA

You probably get a mxArray from your workspace because you need to manage it element by element in C-like style.

KEY FUNCTION:

```
void *mxGetData(const mxArray *pm)
```

(Pointer to the first element of the real data. Returns NULL in C if there is no real data)

In C, `mxGetData` returns a void pointer (void *). Since void pointers point to a value that has no type, cast the return value to the pointer type you are working.

FOCUS ON THIS:

MATLAB stores data in a column-major (columnwise) numbering scheme, which is how Fortran stores matrices. MATLAB uses this convention because it was originally written in Fortran. MATLAB internally stores data elements from the first column first, then data elements from the second column second, and so on, through the last column.

For example, given the matrix:

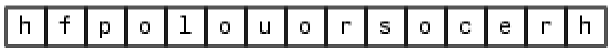
a = ['house'; 'floor'; 'porch']

its dimensions are:

```
size(a)

ans =  3     5
```

and its data is stored as:

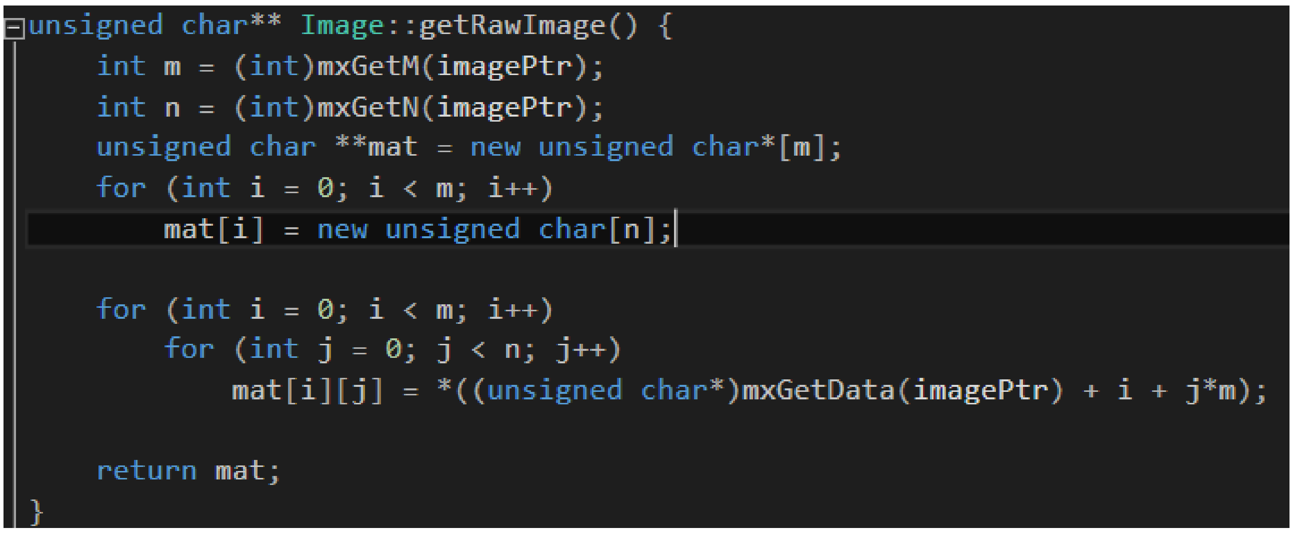


h	f	p	o	l	o	u	o	r	s	o	c	e	r	h

You can understand that you need arithmetic of pointers to move through matrix elements

EXERCISE:
PUTTING A GRAY LEVEL IMAGE EXTRACTED FROM MATLAB ENGINE INTO A C STYLE MATRIX

A SOLUTION:



```cpp
unsigned char** Image::getRawImage() {
    int m = (int)mxGetM(imagePtr);
    int n = (int)mxGetN(imagePtr);
    unsigned char **mat = new unsigned char*[m];
    for (int i = 0; i < m; i++)
        mat[i] = new unsigned char[n];

    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            mat[i][j] = *((unsigned char*)mxGetData(imagePtr) + i + j*m);

    return mat;
}
```

lab_example2.cpp

You can try to implement the reverse action to explore other functions of MATLAB Engine.

This guide is born to give you the first inputs for working with MATLAB in C/C++. Surely it’s not exhaustive and it’s the expression of one way to approch.

Read the documentation for going deeper into the Engine tools.