C++ with Matlab Tutorial

55:148
Digital Image Processing
2007.10.16

Why use C/C++ with Matlab?

- Matlab can be slow
- C++ can be fast
  - How can we integrate C++'s speed with Matlab's convenience?
- Goal:

  -Implement critical functions with (fast) C++ code, and use (slow) Matlab code for data structure management (which can be difficult/annoying in C++)

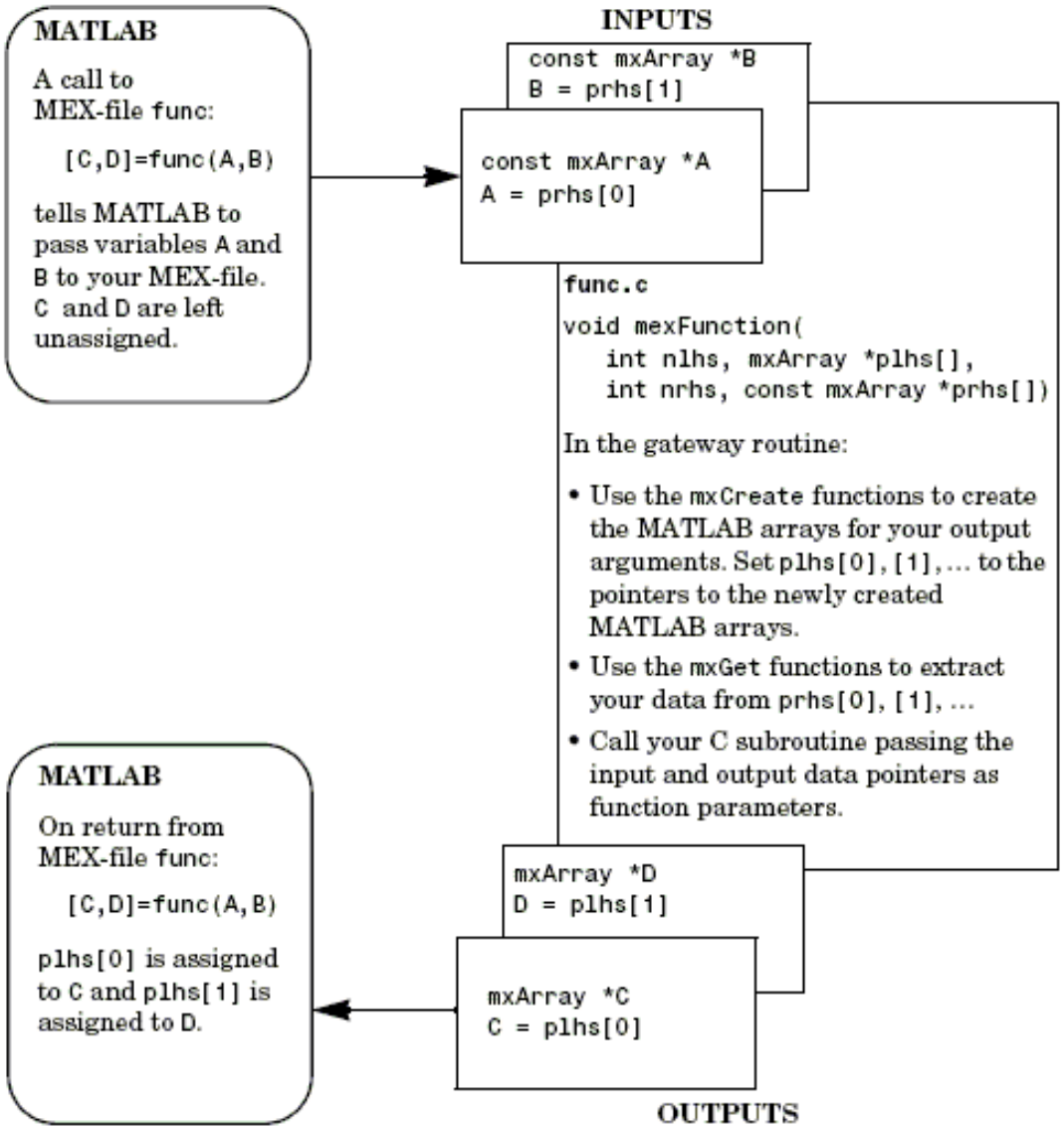
How do we integrate C++ with Matlab?

- Matlab allows the use of MEX files
  - MEX files are pre-compiled files that are called from Matlab
  - Can be compiled from Matlab .m files
  - Can also be compiled from external C++ code.
    - This will be our focus

The components of a MEX file

- Every MEX file consists of two parts:
  - A gateway routine that interfaces with Matlab. This is the entry point for the C++ code. **Must** be called `mexFunction()`
  - Any number of subroutines that are called from the gateway routine. The bulk of your computation will be inside these subroutines.

The components of a MEX file

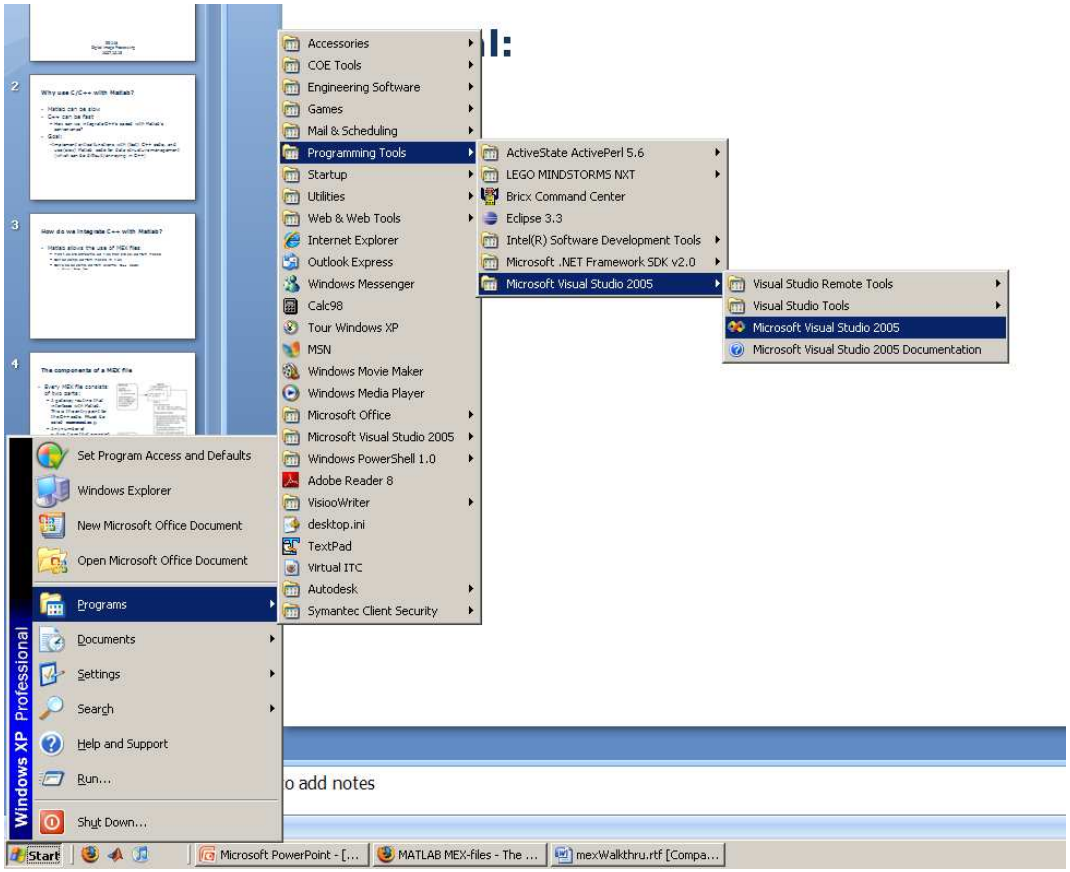


- MEX I/O:
  - The gateway routine is passed four parameters: `int nlhs, mxArray *plhs[], int nrhs, mxArrary *prhs[]`
  - `nlhs`: the number of outputs
  - `plhs[]`: an array of outputs
  - `nrhs`: the number of inputs
  - `prhs[]`: an array of inputs

Tutorial:

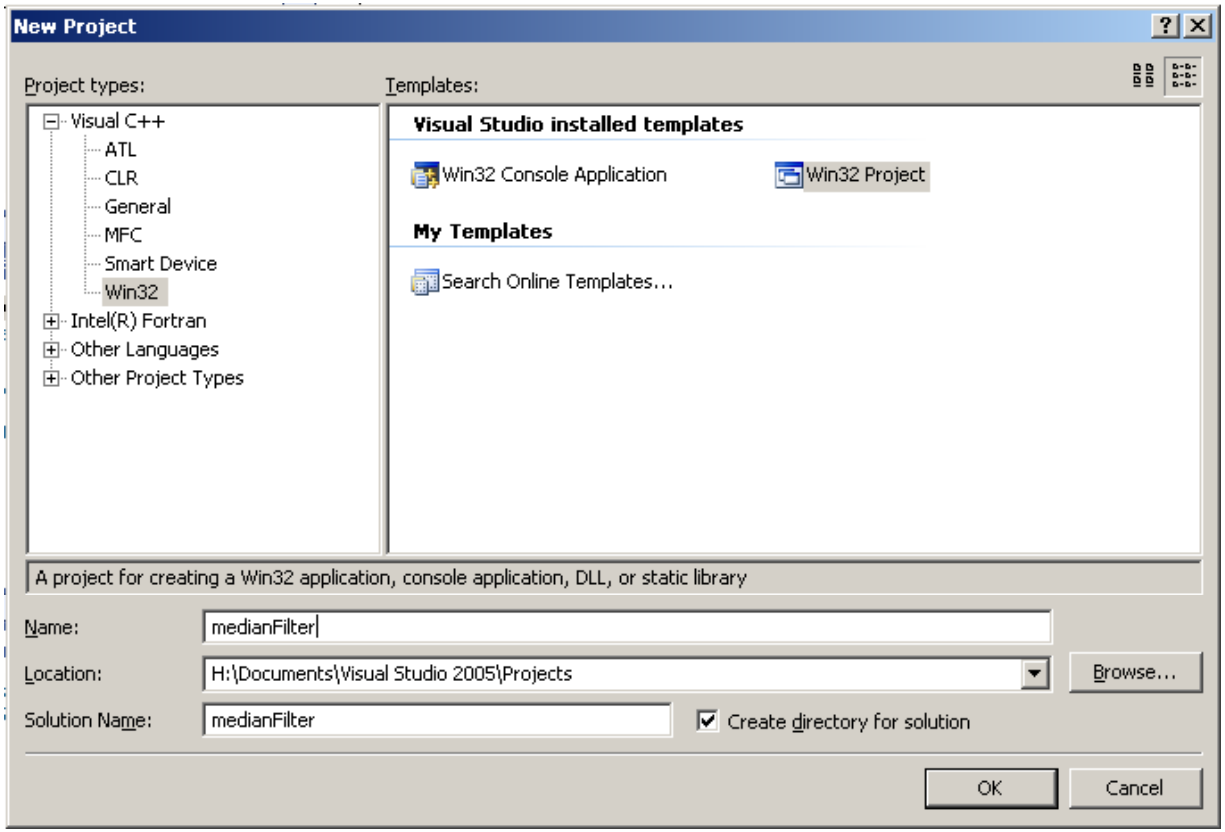
- We will create a MEX file to implement a median filter
- Implemented in C++ using M$ Visual Studio 2005
  - MEX code will be contained in a DLL (Dynamic-Link Library)

Tutorial:



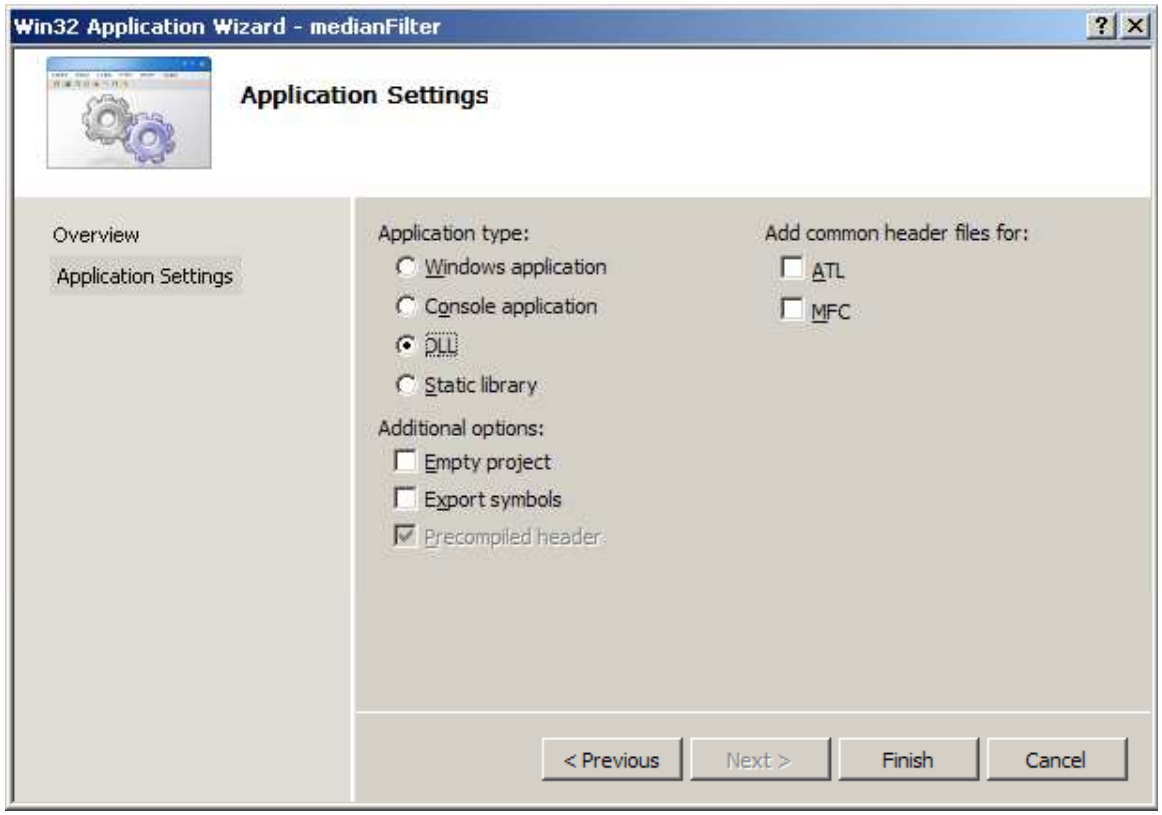
- Open Visual Studio 2005
  - Start->Program Files->Programming Tools ->Microsoft Visual Studio 2005

Tutorial:



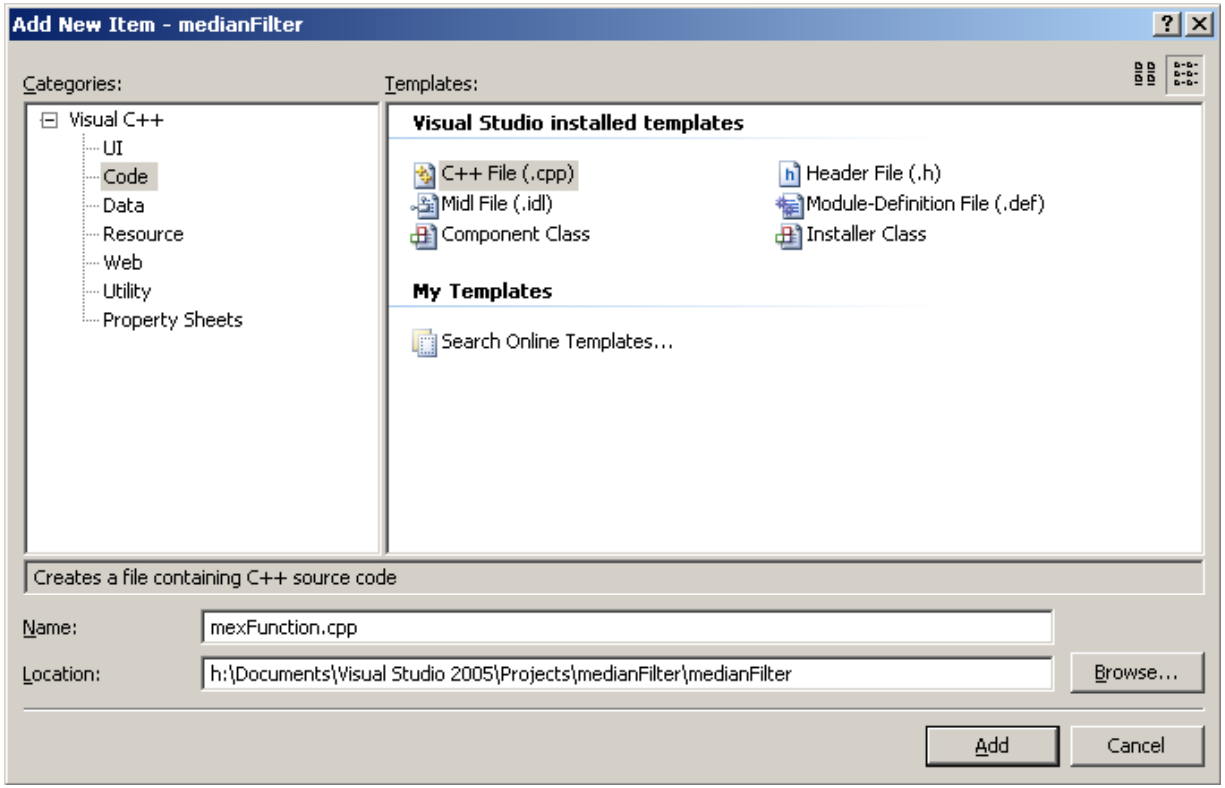
- File->New->Project
  - Select **Visual C++**
  - Select **Win32**
    - select **win32 project**
    - name it MedianFilter
    - **OK**

Tutorial:



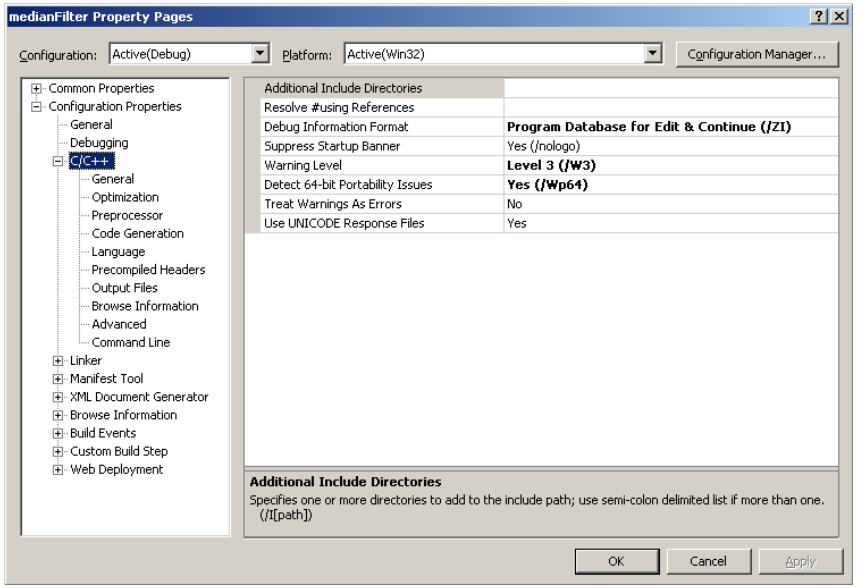
- Click 'Next'
  - Select "**DLL**"
  - Check "**Export Symbols**"
  - Check "**Empty Project**"
  - Click "**Finish**"

Tutorial:



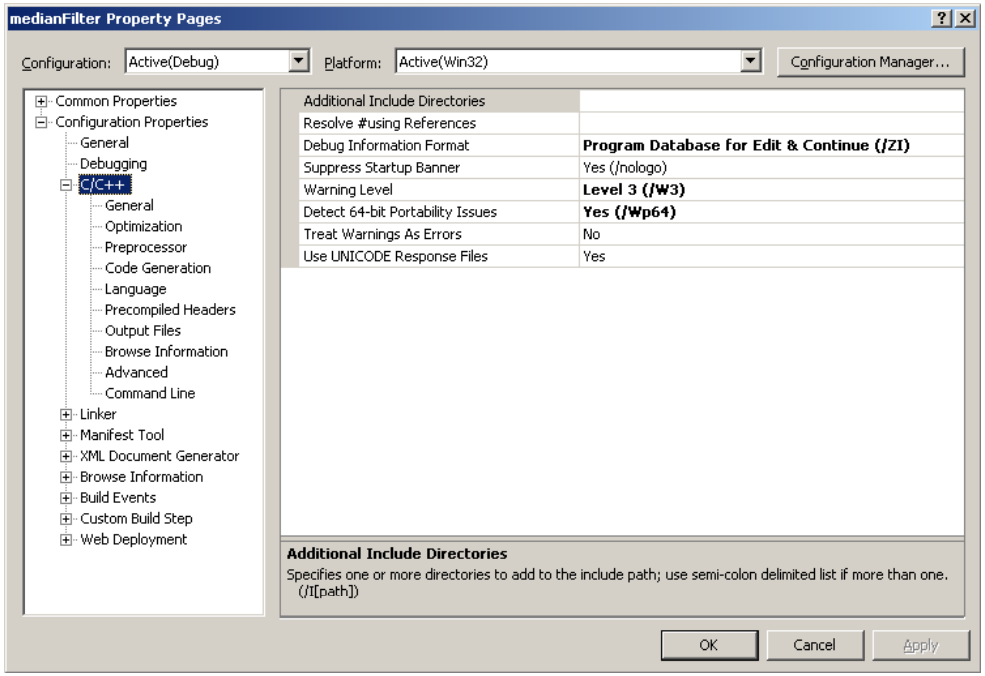
- Project->Add New Item
  - Select "**Code**"
    - Highlight "**C++ File**"
    - Name it "mexFunction.cpp"
    - **Add**

Tutorial:



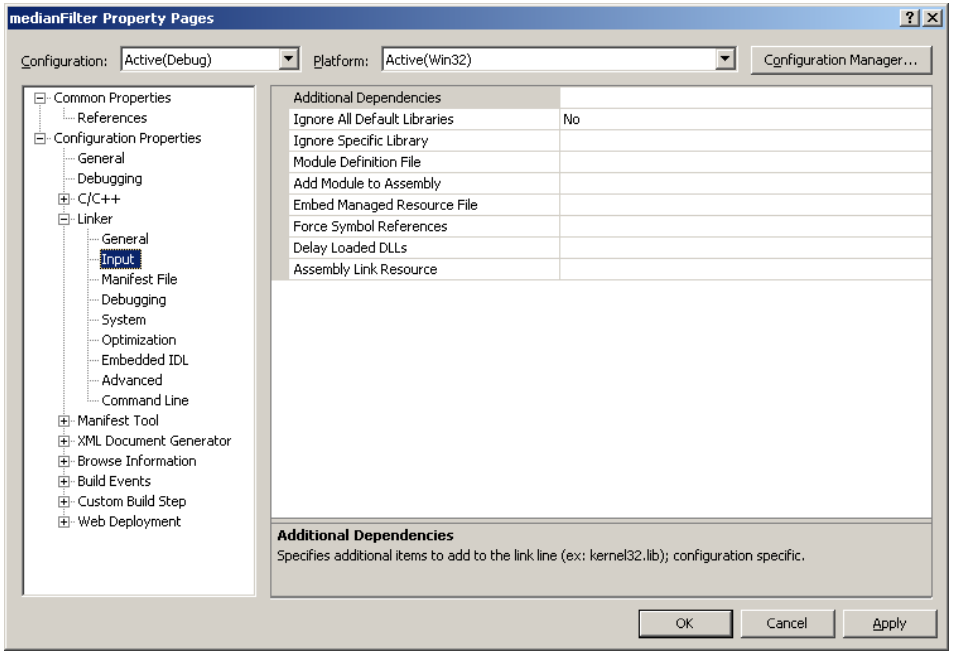
- **Project->MedianFilter Properties**
  - Expand "**Configuration Properties**"
    - Expand "**C/C++**"
      - Highlight "**General**"
        - Click in the "**Additional Include Directories**" field
        - Click the "**...**" button ...
          - Click the "**New Line**" button
          - Click the "**...**" button ...
          - Browse to "C:\Program Files\Mathworks\Matlab\R2007a\extern\include"
          - **OK**

Tutorial:



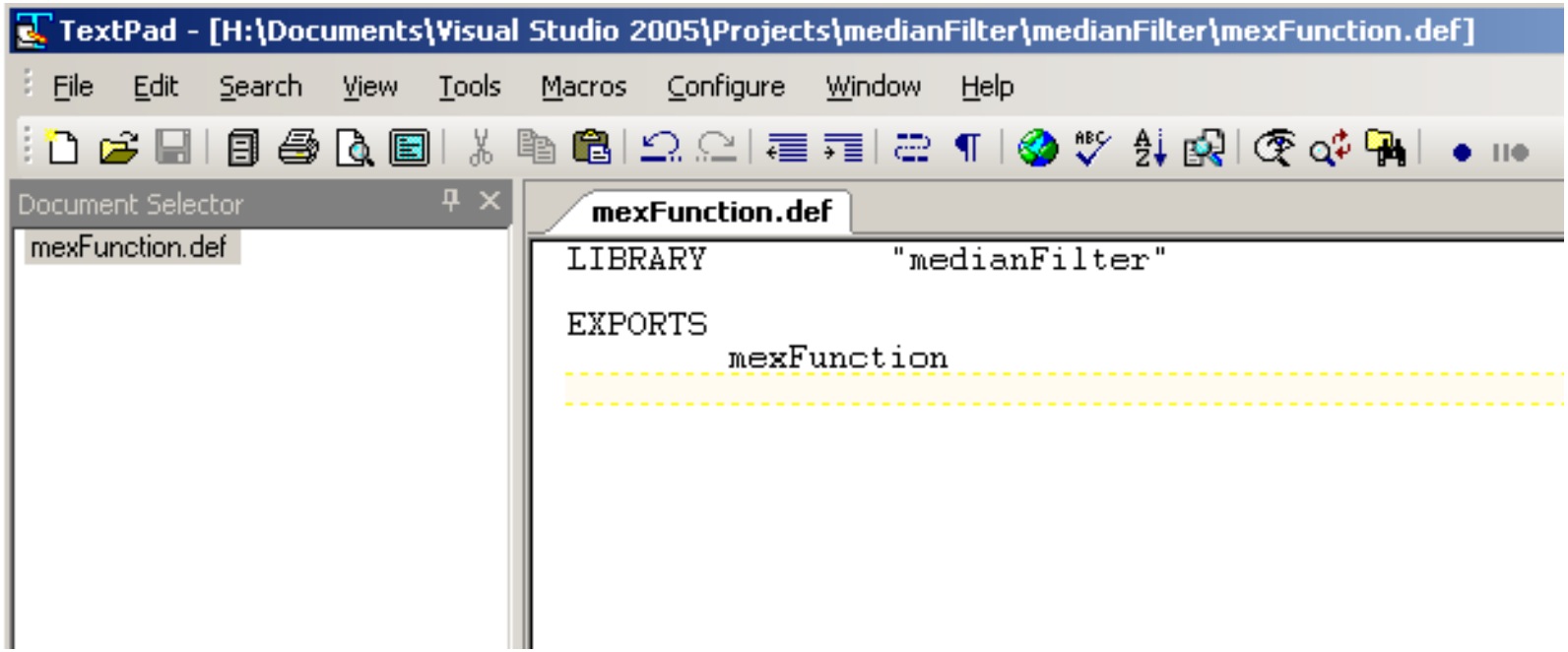
- **Project->MedianFilter Properties (CONTINUED)**
  - Highlight "**Preprocessor**"
    - Click in the "**Preprocessor Definitions**" field
    - Click the "**...**" button
    - Add "MATLAB_MEX_FILE" on a new line
    - **OK**
  - Highlight "**Code Generation**"
    - Change "**Runtime Library**" to "**Multi-threaded Debug**"

Tutorial:



- **Project->MedianFilter Properties (CONTINUED)**
  - Expand "**Linker**"
    - Highlight "**General**"
      - Change "**Output file**" to "$(OutDir)\MedianFilter.dll"
      - In the "**Additional Library Directories**" field, add: "C:\Program Files\ Mathworks\Matlab\R2007a\extern\lib\win32\microsoft"
    - Highlight "**Input**"
      - In "**Module Definition File**", add ".\mexFunction.def"
  - Click **Apply**
  - Click **OK**

Tutorial:



- Add a new text file in your project directory named "mexFunction.def"
  - Add the following lines:

```
LIBRARY     "MedianFilter"
EXPORTS
        mexFunction
```

Tutorial:

- **Project->Add New Item**
  - Select “**Code**”
    - Select “**Header File**”
    - Name it “mexFunction.h”
    - Add
  - Add the following lines to your new header file:

```
#include "matrix.h”
#include "mex.h”
#define MEX_FUNCTION_EXPORTS

#ifdef MEX_FUNCTION_EXPORTS
#define MEX_FUNCTION_API __declspec(dllexport)
#else
#define MEX_FUNCTION_API __declspec(dllimport)
#endif

MEX_FUNCTION_API void mexFunction(int nlhs,
mxArray* plhs[],int nrhs, mxArray* prhs[]);
```

Tutorial:

- In your .cpp file, add the following lines:

```
#include "mexFunction.h"
#pragma comment(lib, "libmx.lib")
#pragma comment(lib, "libmat.lib")
#pragma comment(lib, "libmex.lib")
```

- Check that what you've done so far will compile. Of course, it won't do anything, but it's good to check for errors.

Tutorial:

•We're now in a position to actually start coding. Let's start with our gateway function, `mexFunction()`.

- •Add the following lines to your file:

```
void mexFunction(int nlhs, mxArray*
plhs[], int nrhs, mxArray *prhs[]){

}
```

- •This defines the entry point for our program, and declares the four variables that we will need to talk to Matlab.

Tutorial:

- We should decide what we will be passing in, and what we will be passing back. We're implementing a median filter to process an image, so what will our inputs be? Obviously, we'll need an input image. We could also define an arbitrary filtering radius, but that would makes things a little more complicated. Let's keep it simple: we'll pass a matrix in, and we'll pass the filtered matrix back out.

Tutorial:

- Thus, for our function:

`nlhs = 1`

`nrhs = 1`

`prhs[0]` is the input image,

`plhs[0]` is the output image.

Tutorial:

Where to start? Let's take care of the input matrix first. Define a pointer to a matrix of type `mxArray`:

```
mxArray *matrixIn;
```

Where should this pointer point? Well, how about our input image?

```
matrixIn = prhs[0];
```

Tutorial:

We're working with a 2D array (an image),but it is stored as a 1D array. Thus, it will probably be useful to know where our row and column bounds are. We can get the size of our m$_x$n matrix with:

```
m = mxGetM(matrixIn);
n = mxGetN(matrixIn);
```

Now, the (i,j) element of the image is at:

```
image[(m*i)+j]
```

Tutorial:

Ok. Now we've taken care of the input matrix. Let's define an output matrix of the same size as the input matrix. Remembering that `plhs[0]` is a reference to the output matrix, we can do this with:

```
plhs[0] =
mxCreateDoubleMatrix(m,n,mxREAL);
```

Tutorial:

Now we're in a position to write our `MedianFilter()` function. If we weren't dealing with Matlab, we could call it with:

```
MedianFilter(&matrixIn, &matrixOut, m, n);
```

We're in Matlab-land now, however, so instead of the "address-of" operator (&), we need to do the following:

```
MedianFilter(mxGetPr(matrixIn),
                mxGetPr(plhs[0]), m, n);
```

The `mxGetPr()` function returns the address of the first element of the matrix.

Tutorial:

Let's write our `MedianFilter()` function. We are passing by reference (thus our function doesn't return anything) so it's of type `void`. We are passing it the addresses of two matrices of type `double`, so we declare our first two input arguments as `double*`. We are also passing it two integers, so putting everything together, we have:

Tutorial:

```
void MedianFilter(double * imageIn,
double* imageOut, int m, int n){
}
```

- Make sure you add a function prototype to your header:

```
void MedianFilter(double *,
double*, int, int);
```

Tutorial:

Now for the actual algorithm. we'll begin our loop at three so that we don't need to worry about what to do when our kernel is close to the borders. If we format our input image correctly in Matlab, everything will work out fine…

```
vector<double> neighborhood;
double median;
int u,v,i,j;

//matlab stores matrices by column, so it's
//most efficient to have our outer loop parse
//through rows, and our inner loop parse
//through columns(i.e. *not* //like you're
//reading English)
for(i=3;i<n-2;i++){
   for(j=3;j<m-2;j++){
```

Tutorial:

```
//construct the new neighborhood:
//first, reset the vector
neighborhood.clear();

//get the five horizontal elements
for(u=i-2;u<i+3;u++) {
  neighborhood.push_back(imageIn[(m*u)+j] );
}

//get the two top elements
for(v=j-2;v<j;v++){
  neighborhood.push_back(imageIn[(m*i)+v] );
}

//get the two bottom elements
for(v=j+1;v<j+3;v++){
        neighborhood.push_back(imageIn[(m*i)+v] );
}
```

Tutorial:

```
//we now have a 9 pixel neighborhood...
//first, sort the neighborhood
sort(neighborhood.begin(), neighborhood.end());
//now pick out the median:
median = neighborhood[4];

   //assign the current pixel to the median
   imageOut [(m*i)+j] = median;
```

Tutorial:

Let's see if this compiles. Nope. We forgot to include some things:

```
#include <vector>
#include <algorithm>

using namespace std;
```

Now it should compile. We should probably call our function at some point, so stick this at the bottom of `mexFunction()`:

```
MedianFilter( mxGetPr(matrixIn),
              mxGetPr(plhs[0]), m, n);
```

Tutorial:

Good. We're done with the C++ side of things. Now we need to write the Matlab side of things. Start up Matlab and start a new .m file. This .m file only needs one line:

```
Function filtered =
MedianFilter(I);
```

Tutorial:

Save it as MedianFilter.m in the same directory that Visual Studio wrote the .DLL file (should be "<project directory>\
medianFilter\Debug\"). Now create another .m file, name it test.m or something of the sort.

Tutorial:

Make your test.m file look like this:

```
 %read in a test image
I = imread('medlines.gif');
dim = size(I);   %get the size

%pad the image with 4 pixels
J = zeros(dim+4);
J(3:dim(1)+2,3:dim(2)+2) = I;
```

Tutorial:

```
%take care of padding our extra 4
columns
J(:,1) = J(:,4);
J(:,2) = J(:,3);
J(:,dim(2)+3) = J(:,dim(2)+2);
J(:,dim(2)+4) = J(:,dim(2)+1);

%take care of padding our extra 4 rows
J(1,:) = J(4,:);
J(2,:) = J(3,:);
J(dim(1)+3,:) = J(dim(1),:);
J(dim(1)+4,:) = J(dim(1)-1,:);
```

Tutorial:

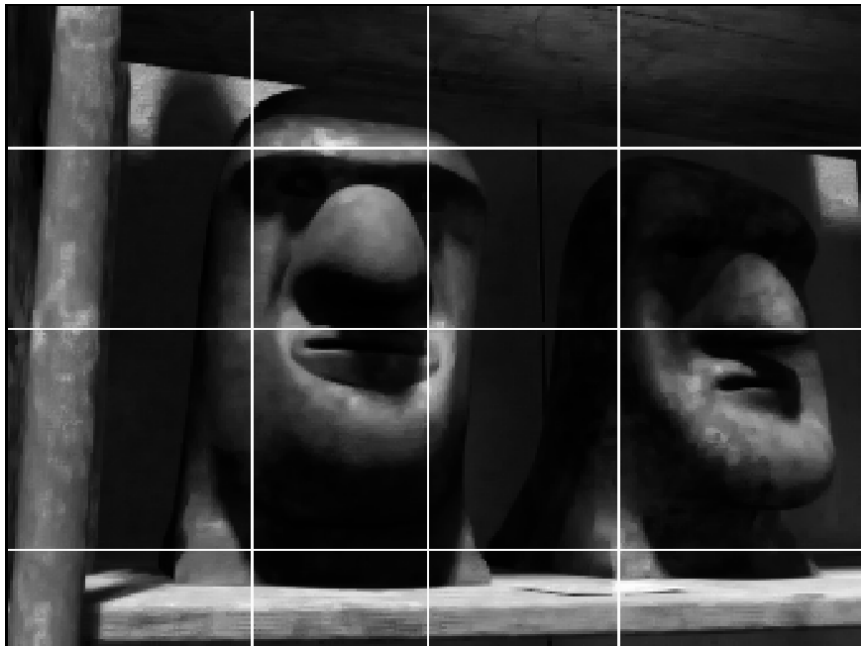
```
%pass our padded image to MEX
routine
K = MedianFilter(J);

%throw away the padding
K = K(3:dim(1)+2,3:dim(2)+2);
imshow(K,[0 255]);          %show it

clear mex    %useful for debugging
```

Tutorial:

Now make sure that your test image is in the same folder as the other files, and run your test.m script. If everything went according to plan, you should see your processed image pop up.



One advantage of this median filter is that it is edge-preserving. If your function worked correctly, you should still see the white lines in the image.

Exercise:

Modify this example to implement a 5x5 averaging filter. Thus, each pixel of the output image should contain the average value of the corresponding 5x5 region of the input image.