Introduction to C and CMex
E177
April 1, 2008
http://jagger.me.berkeley.edu/~pack/e177

Another simple program (array1.c)
This program illustrates array, the increment operator (++), and for loops
#include <stdio.h>
int main(void)
{
    double A[3];
    A[0] = 1.2;
    printf("The value of A is %g.\n",A);
    for (i=0;i<2;i++)
    {
        printf("The value of A[i] is %g.\n",A[i]);
        printf("The address of A[i] is %p.\n",&(A[i]));
        printf("The value of *(A+i) is %g.\n",*(A+i));
    }
    return(0);
}

Dynamic Memory Allocation (allocate1.c)
In this program, memory is allocated while the program runs, not just in variable declarations
#include <stdio.h>
int main(void)
{
    int i, N=3, *Y;
    Y = (int *) calloc(N, sizeof(int));
    *(X+2) = 1.22;
    *(X+1) = 1.21;
    *X = 1.2;
    X = (double *) calloc(N, sizeof(double));
    A[1] = 1.2;
    A[2] = 2.3;
    printf("The value of X is %p.\n",X);
    printf("The value of Y is %p.\n",Y);
    return(0);
}

Functions (subroutines1.c)
Calls to functions pass arguments by value.
#include <stdio.h>
int sub1(int arg1, int *arg2)
{
    int C;
    printf("Before Call\n");
    C = sub1(A,&B);
    printf("After Call\n");
    return(C);
}

A simple program (simple1.c)
This program illustrates the double and pointer types, address operator (&), the dereference operator (*), and the increment operator (++).
#include <stdio.h>
int main(void)
{
    int i;
    double a;
    double *p = &a;
    printf("The value of a is %g, the address is %p.\n",a,&a);
    printf("The address of p is %p.\n",&p);
    printf("The value that p points to is %g.\n",*p);
    return(0);
}

The for loop
The for loop from the previous slide
#include <stdio.h>
int i;
main()
{
    for (i=0; i<2; i++)
    {
        printf("i=%d.\n",i);
        printf("The value of X[i] is %d.\n",X[i]);
        printf("The value of Y is %d.\n",Y);
    }
    return(0);
}

Free memory that was allocated.
#include <stdio.h>
int main(void)
{ int i; int A=2, B=6, C;
    for (i=0; i<2; i++)
    {
        printf("i=%d.\n",i);
        printf("The value of X[i] is %d.\n",X[i]);
        printf("The value of Y is %d.\n",Y);
    }
    return(0);
}

Copyright 2005-8, Andy Packard. This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/2.0/
Matlab variable types: primitives
The three main object classes in Matlab are:
- double
  - multidimensional array of double precision floating point numbers
- char
  - multidimensional array of ascii characters
- logical
  - Multidimensional array of 1-bit (0/1) numbers

Eight other built-in primitive variable types are:
- uint8, uint16, uint32, uint64
  - multidimensional array of 8, 16, 32 or 64-bit, unsigned integers
- int8, int16, int32, int64
  - multidimensional array of 8, 16, 32 or 64-bit, signed integers

Two things to remember:
- Every variable in Matlab is an array
- Arrays are at least 2-dimensional (no notion of 1-d array)

Matlab variable types: derived primitives
Two other important object classes in Matlab are:
- cell
  - multidimensional array of “containers”
- struct
  - multidimensional array of a structure with fields (common across array)

In a cell, the contents of a container may be:
- double, char, intXX, uintYY
- Another cell array
- A struct array
- An object of other classes (listed on next slide)

In a struct, the value of a field may be:
- Same list as above

The mxArray object
Inside the C-programs we write, all of these types of objects will be variables of the type mxArray.
We will never usually have an mxArray variable, but will always have pointers to them. All of the mxArray routines have input arguments that are addresses of mxArray variables.

Typical example within a Cmex file:

```c
mxArray *pMat;
...
if (mxIsComplex(pMat)) { /* code here */ }
el se if (mxIsCell(pMat)) { /* code here */ }
el se if (mxIsStruct(pMat)) { /* code here */ }
```

Remember, in C, the arrayname, by itself, is actually the address of the first element. So, by changing the array elements in the subroutine, you are really changing the array elements back in the calling routine as well...

In this case, the elements of the array pLHS (which are addresses of mxArray objects) will be changed.

The mexFunction gateway (cont’d)

```c
#include "mex.h"
somefunction.c

void mexFunction(int nLHS, mxArray *pLHS[],
                 int nRHS, const mxArray *pRHS[])
{
    /* Code here */
}
```

What happens if in Matlab (ie, at command line, in script, or in function) a command of the form

```matlab
[Y,Z] = somefunction(A,B,C);
```

is executed?

Before executing

```matlab
[Y,Z] = somefunction(A,B,C);
```
we first need to compile the function. At the Matlab prompt, execute the command

```bash
>> mex somefunction.c
```

This will create an executable file in the same folder that can be called directly from Matlab.

You may have to (once) set up the MEX infrastructure if you haven’t already. In that case, type

```bash
>> mex -setup
```
and simply follow the instructions.
The `mexFunction` gateway
Suppose variables `A`, `B` and `C` exist in the current workspace. Executing

```c
[Y,Z] = somefunction(A,B,C)
```

will call the compiled code (function declaration shown below)

```c
void mexFunction(int nLHS, mxArray *pLHS[],
    int nRHS, const mxArray *pRHS[])
```

Upon entry
- `nRHS` will equal 3
- `pRHS[0]` will be the address of mxArray `A`
- `pRHS[1]` will be the address of mxArray `B`
- `pRHS[2]` will be the address of mxArray `C`
- `nLHS` will equal 2
- `pLHS[0]` can hold the address of an mxArray
- `pLHS[1]` can hold the address of an mxArray

The mexFunction gateway

```c
void mexFunction(int nLHS, mxArray *pLHS[], int nRHS, const mxArray *pRHS[])
```

Somefunction.c

```c
double *pReal, *pImag;
mxArray *pMat;
/* code to put a legal address in pMat */
pReal = mxGetPr(pMat);
if (mxIsComplex(pMat)) {
    pImag = mxGetPi(pMat);
}
```

Where is raw data stored in double arrays

If `pMAT` is a pointer to an mxArray, and
- `mxIsDouble(pMat)`
- `mxCreateDoubleScalar` creates a 1-by-1 double

Several `mx`-utilities to create an N-D mxArray
- `mxCreateDoubleArray` creates an N-D double
- `mxCreateCharArray` creates an N-D char

Here is a simple program that returns two arguments.

```c
#include "mex.h"

void mexFunction(int nLHS, mxArray *pLHS[], int nRHS, const mxArray *pRHS[])
{
    if (nLHS==2 && nRHS==0) {
        pLHS[0] = mxCreateString("This is a string");
        pLHS[1] = mxCreateDoubleScalar(11.8);
    }
    else {
        mexErrMsgTxt("Must be: 2 outputs, no inputs.");
    }
}
```

>> [a,b] = createTwo;

Where are the contents of cell arrays

If `pCell` is a pointer to an mxArray, and
- `mxIsCell(pCell)`
- `mxCreateLogicalScalar` creates a 1-by-1 logical

Several `mx`-utilities to create a 2-D mxArray
- `mxCreateDoubleMatrix` creates a 2-d double
- `mxCreateCellMatrix` creates a 2-d cell
- `mxCreateString` creates a 1-by-N char array

Several mx-utilities to create an N-D mxArray
- `mxCreateDoubleArray` creates an N-D double
- `mxCreateCharArray` creates an N-D char
- `mxCreateCellArray` creates an N-D cell
- `mxCreateStructArray` creates an N-D struct

mxCreate utilities

A few special `mx`-utilities to create a scalar mxArray
- `mxCreateLogicalScalar` creates a 1-by-1 logical

Several `mx`-utilities to create a 2-D mxArray
- `mxCreateDoubleMatrix` creates a 2-d double
- `mxCreateCellMatrix` creates a 2-d cell
- `mxCreateString` creates a 1-by-N char array

Several `mx`-utilities to create an N-D mxArray
- `mxCreateDoubleArray` creates an N-D double
- `mxCreateCellArray` creates an N-D cell
- `mxCreateStructArray` creates an N-D struct
- `mxCreateCharArray` creates an N-D char

```c
#include "mex.h"

void mexFunction(int nLHS, mxArray *pLHS[], int nRHS, const mxArray *pRHS[])
{
    if (nLHS==2 && nRHS==0) {
        pLHS[0] = mxCreateString("This is a string");
        pLHS[1] = mxCreateDoubleScalar(11.8);
    }
    else {
        mexErrMsgTxt("Must be: 2 outputs, no inputs.");
    }
}
```

>> [a,b] = createTwo;

```c
#include "mex.h"

void mexFunction(int nLHS, mxArray *pLHS[], int nRHS, const mxArray *pRHS[])
{
    if (nLHS==2 && nRHS==0) {
        pLHS[0] = mxCreateString("This is a string");
        pLHS[1] = mxCreateDoubleScalar(11.8);
    }
    else {
        mexErrMsgTxt("Must be: 2 outputs, no inputs.");
    }
}
```
**Example Vector Cross Product**

Let \( A \) and \( B \) be 3-by-1 column vectors, representing the Cartesian components of vectors. The cross-product of \( A \) and \( B \), denoted \( A \times B \), is also a vector, whose 3 components are given by

\[
A \times B = \begin{bmatrix}
A_1 B_2 - A_2 B_1 \\
A_2 B_3 - A_3 B_2 \\
A_3 B_1 - A_1 B_3
\end{bmatrix}
\]

---

**Vector Cross Product: CMex**

```matlab
function C = cross177m(A,B)
C = zeros(3,1);  
C(1) = A(2)*B(3) - A(3)*B(2);  
C(2) = A(3)*B(1) - A(1)*B(3);  
C(3) = A(1)*B(2) - A(2)*B(1);  
end
```

**CMex with Error Checking**

```c
#include "mex.h"
void mexFunction(int nLHS, mxArray *pLHS[],
int nRHS, const mxArray *pRHS[])
{
  /* Number of Arguments */
  if (nLHS==1 && nRHS==2) {
    /* Both input arguments are DOUBLES */
    if (mxIsDouble(pRHS[0]) && mxIsDouble(pRHS[1])) {
      /* Both input arguments are REAL */
      if (~mxIsComplex(pRHS[0]) && ~mxIsComplex(pRHS[1])) {
        /* 1st input argument should be 2-d */
        if (mxGetNumberOfDimensions(pRHS[0])==2) {
          /* 2nd input argument should be 2-d */
          if (mxGetNumberOfDimensions(pRHS[1])==2) {
            /* Both input arguments need 3 rows */
            if (mxGetM(pRHS[0])==3 && mxGetM(pRHS[1])==3) {
              /* Both input arguments need 1 column */
              if (mxGetN(pRHS[0])==1 && mxGetN(pRHS[1])==1) {
                ...  
              }  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

**Calling Matlab functions in CMex file**

```c
#include "mex.h"
void mexFunction(int nLHS, mxArray *pLHS[],
int nRHS, const mxArray *pRHS[])
{
  mxArray *localIN[4], *localOUT[3];
  int rval;
  localIN[0] = pRHS[0];
  rval = mexCallMATLAB(2,localOUT,1,localIN,"eig");
  pLHS[0] = localOUT[1];
  mxDestroyArray(localOUT[0]);
}
```