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

Variables and Addresses

```c
int B=7;
double A=3.2, D;
double *P;   /* addressofdouble P */
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Contents@Address (ie. value of variable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0xFA12</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>0xFA32</td>
<td>3.2</td>
</tr>
<tr>
<td>D</td>
<td>0xFA4A</td>
<td>garbage</td>
</tr>
<tr>
<td>P</td>
<td>0xFA70</td>
<td>garbage</td>
</tr>
</tbody>
</table>

P = &A;

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Contents@Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0xFA12</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>0xFA32</td>
<td>4.71</td>
</tr>
<tr>
<td>D</td>
<td>0xFA4A</td>
<td>13.9</td>
</tr>
<tr>
<td>P</td>
<td>0xFA70</td>
<td>0xFA32</td>
</tr>
</tbody>
</table>

*P = 4.71; D = 13.9;

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Contents@Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0xFA12</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>0xFA32</td>
<td>4.71</td>
</tr>
<tr>
<td>D</td>
<td>0xFA4A</td>
<td>13.9</td>
</tr>
<tr>
<td>P</td>
<td>0xFA70</td>
<td>0xFA32</td>
</tr>
</tbody>
</table>

P = &D;

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Contents@Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0xFA12</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>0xFA32</td>
<td>4.71</td>
</tr>
<tr>
<td>D</td>
<td>0xFA4A</td>
<td>13.9</td>
</tr>
<tr>
<td>P</td>
<td>0xFA70</td>
<td>0xFA4A</td>
</tr>
</tbody>
</table>

A simple program (simple1.c)

This program illustrates the `double` and `pointer` variable types, address operator (&), the dereference operator (*) and the `printf` output function.

#include <stdio.h>
int main(void)
{
    int b;
    double a;
    double *p;  /* Defines a variable p */
    /* The variable p can hold the address of a DOUBLE */
    /* This does not declare a DOUBLE */
    a = 1.3;
    b = -6;
    printf("The value of a and b are %g, %d\n",a,b);
    printf("The addresses of a and b are %p, %p\n",&a,&b);
    p = &a;   /* & is address operator */
    printf("The value of p is %p\n",p);
    printf("The address of p is %p\n",&p);
    printf("The value that p points to is %g\n",*p);
    a = 2.3;
    printf("The value of a is %g, the address is %p\n",a,&a);
    printf("The value that p points to is %g\n",*p);
    return(0);
}

Another simple program (array1.c)

This program illustrates arrays, the increment operator (++) and `for` loops

```c
#include <stdio.h>
int main(void)
{
    double A[3];
    int i;
    A[0] = 1.2;
    A[1] = 2.2;
    A[2] = 3.2;
    printf("The value of A is %p\n",A);
    for (i=0;i<=2;i++) {
        printf("i=%d\n",i);
        printf("The value of A[%d] is %g\n",i,A[i]);
        printf("The value of *(A+%d) is %g\n",i,*(A+i));
        printf("The value of A+%d is %p\n",i,A+i);
        printf("The address of A[%d] is %p\n",i,&(A[i]));
    }
    return(0);
}
```

Dynamic Memory Allocation (allocate1.c)

In this program, memory is allocated while the program runs, not just in variable declarations

```c
#include <stdio.h>
int main(void)
{
    double *X;
    int i, N=3, *Y;
    X = (double *) calloc(N, sizeof(double));
    *X = 1.2;
    *(X+1) = 1.21;
    *(X+2) = 1.22;
    Y = (int *) calloc(N, sizeof(int));
    printf("The address of X is %p\n",&X);
    printf("  The value of X is %p\n",X);
    printf(" The value of *X is %g\n",*X);
    for (i=0;i<2;i++) {
        printf("i=%d\n",i);
        printf("The value of X[%d] is %g\n",i,X[i]);
        printf("The value of *(X+%d) is %g\n",i,*(X+i));
        printf("The value of X+%d is %p\n",i,X+i);
        printf("The value of Y+%d is %p\n",i,Y+i);
        printf("The address of X[%d] is %p\n",i,&(X[i]));
    }
    free(X);
    free(Y);
    return(0);
}
```
Two things to remember

Every variable in Matlab is an array

Arrays are at least 2-dimensional (no notion of 1-d array)

and...

Recall from Lecture #1

Matlab variable types: primitives

The three main object classes in Matlab are:

- double
- char
- logical

Eight other built-in primitive variable types are:

- uint8, uint16, uint32, uint64
- logical
- cell
- struct

Two other important object classes in Matlab are:

- cell
- struct

Recall, from Lecture #1

Matlab variable types: derived primitives

Two other important object classes in Matlab are:

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

Typical example within a CMex file:

```c
mxArray *pMat;
... /* code to put a legal address in pMat */
if (mxIsComplex(pMat)) { /* code here */
else if (mxIsCell(pMat)) { /* code here */
else if (mxIsStruct(pMat)) { /* code here */
```

The mxArray object (cont'd)
TRUE if pMat points to a struct, FALSE otherwise.

Structures in C
Declaring a structure in C involves tagging/identifying its type, and listing the fields. For example, we might declare structures of type matrix as

```c
struct matrix
{
    char *name; /* address to first character */
    int ndims; /* number of dimensions */
    int *dims; /* address to dimensions */
    double *addRealPart; /* address to real part */
    double *addImagPart; /* address to imag part */
};
```

```c
int D;
struct matrix A, B, *C;
A.addRealPart = (double *) calloc(10, sizeof(double));
C = &A; (*C).ndims = 3; C->ndims = 3;
```

The mexFunction gateway
Suppose we have a file somefunction.c

```c
#include "mex.h"
void mexFunction(int nLHS, mxArray *pLHS[],
int nRHS, const mxArray *pRHS[])
{
    /* Code here */
}
```

The mexFunction gateway (cont'd)
somefunction.c looks like

```c
#include "mex.h"
void mexFunction(int nLHS, mxArray *pLHS[],
int nRHS, const mxArray *pRHS[])
{
    /* Code here */
}
```

The mexFunction gateway (cont'd)
The function definition says
- mexFunction returns nothing (void)
- mexFunction accepts 4 arguments
  - 1st argument is an integer
  - 2nd argument is an array of (pointers to mxArray)
  - 3rd argument is an integer
  - 4th argument is an array of (pointers to mxArray)
- moreover, the value of this variable will not be changed (const qualifier)

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
Where is raw data stored in double arrays

If pMAT is a pointer to an mxArray, and
mxIsDouble(pMat)
is TRUE, then
- mxGetPr returns the address of the first element of the real part
  (stored in linear order)
- mxGetPi returns the address of the first element of the imaginary part

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

Where are the contents of cell arrays

If pCell is a pointer to an mxArray, and
mxIsCell(pCell)
is TRUE, then with integer J ≥ 0,
- mxGetCell(pCell,J) returns the address of the (J+1)th mxArray in the cell (using linear ordering)

```c
int idx;
mxArray *pMat, *pCell;
...
idx = 0; /* the first element */
pMat = mxGetCell(pCell,idx);
idx = 1;
pMat = mxGetCell(pCell,idx); /* 2nd elmnt */
```

Let's use this to study addresses in cells, in pdisplay.c

### mx2ZZZ and mex2ZZZ utilities in pdisplay.c

- mxIsCell
- mexPrintf
- mxGetNumberOfDimensions
- mxGetM
- mxGetN
- mxIsDouble
- mxIsStruct
- mexErrMsgTxt

### Help for mx2ZZZ and mex2ZZZ utilities

- `mxGetNumberofDimensions`: Returns the number of dimensions of an mxArray.
- `mxGetM` and `mxGetN`: Return the number of rows and columns of a 2-D mxArray, respectively.
- `mxIsDouble` and `mxIsLogical`: Check if an mxArray is a double or logical type.
- `mexPrintf`: Prints formatted messages.
- `mexErrMsgTxt`: Prints an error message.

### mxCreate utilities

A few special `mx`-utilities to create a scalar `mxArray`
- `mxCreateDoubleScalar`: Creates a 1-by-1 double mxArray.
- `mxCreateLogicalScalar`: Creates a 1-by-1 logical mxArray.

Several `mx`-utilities create a 2-D `mxArray`
- `mxCreateDoubleMatrix`: Creates a 2-d double mxArray.
- `mxCreateCellMatrix`: Creates a 2-d cell mxArray.
- `mxCreateStructMatrix`: Creates a 2-d structure mxArray.
- `mxCreateString`: Creates a 1-by-N char mxArray.

Several `mx`-utilities to create an N-D `mxArray`
- `mxCreateDoubleArray`: Creates an N-D double mxArray.
- `mxCreateCellArray`: Creates an N-D cell mxArray.
- `mxCreateStructArray`: Creates an N-D structure mxArray.
- `mxCreateCharArray`: Creates an N-D char mxArray.

### mxCreate utilities

The `mxCreate` utilities return a pointer to an `mxArray`.

```c
mxArray *pArray, *pArray2;
...
pArray = mxCreateDoubleScalar(4.3);
pArray2 = mxCreateString("A string");
```

This is one way to create output arguments. Recall for the `mex` gateway,

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

- `pLHS[0]` can hold the address of an mxArray
- `pLHS[1]` can hold the address of an mxArray
- ...
Here is a simple program that returns two arguments.

```c
#include "mex.h"
void maxFunction(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;

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×B, is also a vector, whose 3 components are given by

\[
A \times B = \begin{bmatrix}
A_2B_3 - A_3B_2 \\
A_3B_1 - A_1B_3 \\
A_1B_2 - A_2B_1
\end{bmatrix}
\]

```matlab
cross177m.m
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);
```

```
cross177c.c
#include "mex.h"
void mexFunction(int nLHS, mxArray *pLHS[], int nRHS, const mxArray *pRHS[])
{ double *pA, *pB, *pC;
  pLHS[0] = mxCreateDoubleMatrix(3,1,mxREAL);
  pC = mxGetPr(pLHS[0]);  /* 3-by-1 answer starts here */
  pA = mxGetPr(pRHS[0]);  /* pointer to data in A */
  pB = mxGetPr(pRHS[1]);  /* pointer to data in B */
}
```

Calling Matlab functions in CMex file
Follow the `mexFunction` gateway syntax

```
#include "mex.h"
void maxFunction(int nLHS, mxArray *pLHS[], int nRHS, const mxArray *pRHS[])
{ double *pa, *pb, *pc;
  mxArray *localIN[4], *localOUT[3];
  int rval;
  localIN[0] = ...;
  ... localIN[3] = ...;
  rval = mexCallMATLAB(3,localOUT,4,localIN,"fname");
}
```