

Introduction to C and CMex

E177

April 1, 2008

<http://jagger.me.berkeley.edu/~pack/e177>

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/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94306, USA.

Variables and Addresses

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

Name	Address	Contents@Address (ie. value of variable)
B	0xFA12	7
A	0xFA32	3.2
D	0xFA4A	garbage
P	0xFA70	garbage

P = &A;

Name	Address	Contents@Address
B	0xFA12	7
A	0xFA32	3.2
D	0xFA4A	garbage
P	0xFA70	0xFA32

*P = &D;

Name	Address	Contents@Address	Name	Address	Contents@Address
B	0xFA12	7	B	0xFA12	7
A	0xFA32	4.71	A	0xFA32	4.71
D	0xFA4A	13.9	D	0xFA4A	13.9
P	0xFA70	0xFA32	P	0xFA70	0xFA4A

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);
}
```

Variable declarations

addressofdouble p;

Format specifiers

Assign the value of p to be the address of a

Dereference operator: *p is the value at memory address stored in p

Change the value of the variable a. Remember that p still contains the address of a

Another simple program (array1.c)

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

```
#include <stdio.h>
int main(void)
{
    double A[3];
    int i;
    A[0] = 1.2;
    A[1] = 1.21;
    A[2] = 1.22;
    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);
}
```

Declare A to be an array of 3 doubles

Assign values to the elements of A

By definition, the "value" of the array, by itself, is the address of its first element.

The for loop

The for loop from the previous slide

```
int i;
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]);
}
```

Initialisation, done once before the loop is entered

TEST: if TRUE, body is executed. Loop terminates when condition is false

repeat

Executed after Body is executed (almost always an increment of some sort)

Start of BODY

End of BODY

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)
{
    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);
}
```

Returned value (X) is the address of the beginning of the allocated memory

Allocate memory to hold N elements whose size is the same as DOUBLES

Assign values at the addresses X, X+1 and X+2.

Allocate memory to hold N INTS

Free memory that was dynamically allocated but no longer used.

Functions (subroutines1.c)

Function calls pass arguments by value (a copy is passed into subroutine)

```
#include <stdio.h>
int sub1(int arg1, int *arg2);

int main(void)
{
    int A=2, B=6, C;
    printf("Before Call\n");
    printf(" A = %d, A_Address = %p.\n", A, &A);
    printf(" B = %d, B_Address = %p.\n", B, &B);
    printf(" C = %d, C_Address = %p.\n", C, &C);
    C = sub1(A,&B);
    printf("After Call\n");
    printf(" A = %d, A_Address = %p.\n", A, &A);
    printf(" B = %d, B_Address = %p.\n", B, &B);
    printf(" C = %d, C_Address = %p.\n", C, &C);
    return(0);
}
```

Subroutines pass arguments by their value (i.e., a copy). If you want to change the input arguments, don't pass the variables, pass their addresses.

Similarly, if the variable takes up a lot of memory, pass its address (small)

```
int sub1(int a1, int *a2)
{
    printf(" IN SUB1, BEFORE calculation.\n");
    printf(" a1=%d, address=%p.\n", a1, &a1);
    printf(" a2=%p, address=%p.\n", a2, &a2);
    a1 = a1 + 1;
    *a2 = *a2 + 3;
    printf(" IN SUB1, AFTER calculation.\n");
    printf(" a1=%d, address=%p.\n", a1, &a1);
    printf(" a2=%p, address=%p.\n", a2, &a2);
    return(a1+*a2);
}
```

Functions (subroutines2.c)

Function calls pass arguments by value (a copy is passed into subroutine). Recall that the "value" of an array is the address of its 1st element.

```
#include <stdio.h>
void sub2(int arg1());

int main(void)
{
    int A[2];
    A[0] = 2.0; A[1] = 4.0;
    printf("Before Call\n");
    printf(" A[0] = %d\n", A[0]);
    printf(" A[1] = %d\n", A[1]);
    printf(" A = %p\n", A);
    sub2(A);
    printf("After Call\n");
    printf(" A[0] = %d\n", A[0]);
    printf(" A[1] = %d\n", A[1]);
    printf(" A = %p\n", A);
    return(0);
}
```

Subroutines pass arguments by their value (i.e., a copy). The "value" of an array is the address of its first element. So, passing an array to a subroutine still allows for the subroutine to change the contents of the array.

```
void sub2(int a1[])
{
    printf(" IN SUB2, BEFORE calculation.\n");
    printf(" a1=%p\n",a1);
    printf(" a1[0]=%d\n",a1[0]);
    printf(" a1[1]=%d\n",a1[1]);
    a1[0] = a1[0] + a1[1];
    printf(" IN SUB2, AFTER calculation.\n");
    printf(" a1=%p\n",a1);
    printf(" a1[0]=%d\n",a1[0]);
    printf(" a1[1]=%d\n",a1[1]);
}
```

Functions (subroutines3.c)

Is there a difference between
 - a double (or int, etc), and
 - an array (of length 1) of doubles (or ints, etc)?

Both can store one double precision number.

```
#include <stdio.h>
void sub3(double arg1, double arg2[]);

int main(void)
{
    double A, B[1];
    A = 2.0; B[0] = 4.0;
    printf("Before Call\n");
    printf(" A = %g\n", A);
    printf(" B[0] = %g\n", B[0]);
    sub3(A,B);
    printf("After Call\n");
    printf(" A = %g\n", A);
    printf(" B[0] = %g\n", B[0]);
    return(0);
}
```

Answer: As expected, they get passed differently. Everything gets passed by value, but the "value of an array," is defined as the address of its first element.

```
void sub3(double a1, double a2[])
{
    printf(" IN SUB3, BEFORE calculation.\n");
    printf(" a1=%g\n",a1);
    printf(" a2[0]=%g\n",a2[0]);
    a1 = 7.0;
    a2[0] = a2[0] + 9.0;
    printf(" IN SUB3, AFTER calculation.\n");
    printf(" a1=%g\n",a1);
    printf(" a2[0]=%g\n",a2[0]);
}
```

MasterMind example

```
Code [C0 C1 C2 C3]
Guess [G0 G1 G2 G3]
Look [ 1 1 1 1 ]
rcrp right-color/right-place
umg, unmatched guesses
ui, unmatched counter
rcwp right-color/wrong-place
```

```
#include "mex.h"
void mexFunction(int nlhs, mxArray *pLHS[],
                 int nrhs, const mxArray *pRHS[])
{
    int look[4], umg[4], icode[4], iguess[4];
    int ui, i, j, go, rcrp, rcwp;
    double *pCode, *pGuess, *pout;
    pCode = mxGetPr(pRHS[0]);
    pGuess = mxGetPr(pRHS[1]);

    rcrp = 0; rcwp = 0; ui = 0;
    for (i=0;i<4;i++) {
        icode[i] = (int) pCode[i];
        iguess[i] = (int) pGuess[i];
        look[i] = 1;
        if (icode[i]==iguess[i]) {
            look[i] = 0;
            rcrp++;
        } else { umg[ui] = iguess[i]; ui++; }
    }

    for (i=0;i<4;i++) {
        j = 0; go = 1;
        while (j<4 && go) {
            if (umg[j]==icode[i] && look[j]==1) {
                rcrp++;
                look[j] = 0; go = 0;
            } else { j++; }
        }
    }
    pLHS[0] = mxCreateDoubleMatrix(1,2,MX_REAL);
    pOut = mxGetPr(pLHS[0]);
    pOut[0] = (double) rcrp;
    pOut[1] = (double) rcwp;
}
```

Loop through unmatched guesses

Compare to all codes that have not been matched yet (initial loop, or earlier here). These have look=1

Create 1-by-2 Double mxArray

Get pointer to real part, fill

Recall, from Lecture #1

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)

and...

also, from Lecture #1

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)

Managed arrays of pointers to Matlab variables

In a **struct**, the value of a field may be

- Same list as above

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 `mxZZZZ` routines have input arguments that are addresses of `mxArray` variables.

Typical example within a CMex file

```
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 */ };
```

pMat can hold the address of an mxArray

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

```
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 */
};
```

```
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`

```
#include "mex.h"
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

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

is executed?

The mexFunction gateway (cont'd)

`somefunction.c` looks like

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

The function definition says

- mexFunction returns nothing (void)
- mexFunction accepts 4 arguments

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... See subroutines2.c example.

- 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

Before executing

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

we first need to compile the function. At the Matlab prompt, execute the command

```
>> 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

```
>> mex -setup
```

and simply follow the instructions.

The mexFunction gateway

Suppose variables `A`, `B` and `C` exist in the current workspace.

Executing

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

will call the compiled code (function declaration shown below)

```
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

mexFunction Code needs to

- create 2 mxArrays
- fill them in with results, and
- put their addresses in pLHS

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

```
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)

cell is basically an array of pointers to `mxArray` objects

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

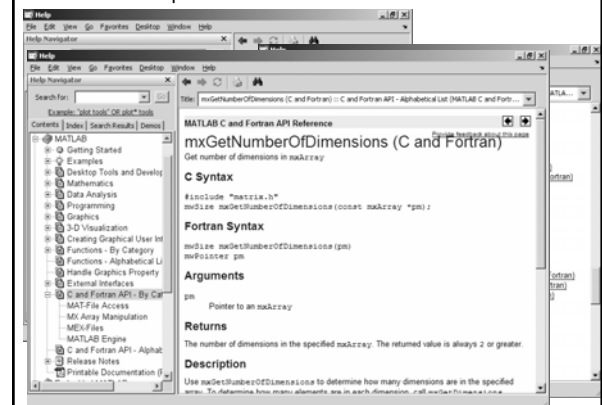
Let's use this to study addresses in cells, in `pdisplay.c`

mxZZZ and mexZZZ utilities in pdisplay.c

```
mxIsCell
mxPrintf
mxGetNumberOfDimensions
mxGetM
mxGetN
mxIsDouble
mxIsStruct

mexErrMsgTxt
```

Help for mxZZZ and mexZZZ utilities



mxCreate utilities

A few special `mx`-utilities to create a scalar `mxArray`

- `mxCreateDoubleScalar` creates a 1-by-1 double
- `mxCreateLogicalScalar` creates a 1-by-1 logical

Several `mx`-utilities create a 2-D `mxArray`

- `mxCreateDoubleMatrix` creates a 2-d double
- `mxCreateCellMatrix` creates a 2-d cell
- `mxCreateStructMatrix` creates a 2-d struct
- `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

mxCreate utilities

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

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

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

```
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`
- ...

`mexFunction` Code needs to

- create `mxArrays`
- fill them in with results, and
- put their addresses in `pLHS`

mxCreate utilities

Here is a simple program that returns two arguments.

```
#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.");
    }
}
```

createTwo.c

>> [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 $A \times B$, is also a vector, whose 3 components are given by

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

```
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);
```

cross177m.m

Need some error checking too...

Vector Cross Product : CMex

```
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);
```

cross177m.m

```
#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 */
  pC[0] = pA[1]*pB[2] - pA[2]*pB[1];
  pC[1] = pA[2]*pB[0] - pA[0]*pB[2];
  pC[2] = pA[0]*pB[1] - pA[1]*pB[0];
}
```

cross177c.c

Need some error checking too...

CMex with Error Checking

```
void mexFunction(int nLHS, mxArray *pLHS[],
                 int nRHS, const mxArray *pRHS[])
{ double *pA, *pB, *pC;
  /* 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) {
```

cross177cAll.c

Calling Matlab functions in CMex file

Follow the `mexFunction` gateway syntax

- Declare two arrays that hold addresses of `mxArrays`
- Fill one array with addresses of `mxArray` (the input arguments)
- Call `mexCallMATLAB`, passing it 5 arguments
 - Number of output arguments
 - (unfilled) array to hold output argument addresses
 - Number of input arguments
 - (filled) array holding the input argument addresses
 - Matlab function name

```
mxArray *localIN[4], *localOUT[3];
int rval;

localIN[0] = ...;
...
localIN[3] = ...;

rval = mexCallMATLAB(3,localOUT,4,localIN,"fname");
```

Calling Matlab functions in CMex file

```
#include "mex.h"
void mexFunction(int nLHS, mxArray *pLHS[],
                 int nRHS, const mxArray *pRHS[])
{
    mxArray *localIN[1], *localOUT[2];
    int rval;

    localIN[0] = pRHS[0];

    rval = mexCallMATLAB(2,localOUT,1,localIN,"eig");

    pLHS[0] = localOUT[1];

    mxDestroyArray(localOUT[0]);
}
```

Destroy unused mxArrays

callEig.c