C – Memory and Pointers

Paul Schrimpf

January 16, 2009

"Other advanced languages, such as assembler and C, were not terribly complex in themselves, but the environments in which applications were developed were downright weird, with mines scattered about everywhere, ready to blow the inattentive programmer out of the water." – Bruce Tognazzini

Pointers and Memory – Introduction

- In C, you must manage memory yourself
  - Allocate and deallocate arrays whose size is only known at run-time
- Pointers give you direct access to memory

Pointers

```
1  double *p; /* p is a pointer to a double */
2  double x,y[10];
3
4  p = &x;    /* p = address of x */
5  (*p) = 1; /* makes x=1 */
6
7  /* pointer arithmetirc */
8  p = &y[0];
9  for(p=&y[0]; p<&y[10]; p++) (*p) = 3; /* makes y[0], y[1], ..
10
11 /* pointer problems */
12 p = &x + 10;  /* p points to 10*sizeof(double) bytes after &x
13 x = (*p);  /* undefined behavior, crashes if you're lucky */
```

Arrays are Pointers

- An array is just a const pointer with memory already allocated
- Multidimensional array are pointers to pointers to ... to the base type

```
1  double x[2], **p;
2  p = x;  /* same as p=&x[0] */
3  /* now, p[0]=x[0], p[1]=x[1], etc */
4  if (x[1]==(*(x+1))) printf(``This is always true'');
5  /* the left side is more readable, the right side is more
6     about what the computer does */
7  /* can index pointers like arrays */
8  p[3] = 5;  /* does something bad */
9
10 /* cannot do arithmetic on array */
11 x++; /* not allowed */
```

Functions of Pointers

- The only way to pass an array to a function is with a pointer, the following are equivalent

```
1 double f(double x[2]);  /* The 2 does nothing */
2 double f(double x[]);
3 double f(double *x);
```

- The only way to return an array is with a pointer

```
1 void vecMult(double out[], double x[], double y[], int n)
2   int i;
3   for(i=0;i<n;i++) out[i] = x[i]*y[i];
4 }
```

A Subtle Error

This code will not work – do you see why?

```
1  #define MAX 1000;
2  double *vecMult2(double x[], double y[], int n) {
3    double out[MAX];
4    int i;
5    if (n>MAX) {
6      printf(``ERROR: vecMult() maximum array size exceeded\n''
7      exit(-1);
8    }
9    for(i=0;i<n;i++) out[i] = x[i]*y[i];
10   return(&out[0]);
11 }
```

Memory Operations

```
1  #include <stdlib.h> /* contains memory functions */
2  double *p, *p2;
3  int **ip, i, N=5;
4
5  p = malloc(sizeof(double)*10); /* allocate room for 10 double
6  p2 = calloc(555, sizeof(double)); /* allocate 555 doubles and
7                                       zero */
8  realloc(&p,sizeof(double)*100); /* changes p to have room for
9                                     doubles, first 10 will sta
10                                    same */
11 ip = malloc(sizeof(int*)*N); /* N pointers to int */
12 for(i=0;i<N;i++) ip[i] = calloc(3,sizeof(int));
13
14 free(p);  /* deallocates memory pointed to by p */
```

- Memory allocated in this way is persistent – it stays allocated even after a function exits

vecMult() – again

```
1  double *vecMult3(double x[], double y[], int n) {
2    double *out;
3    int i;
4    if !(out=malloc(n*sizeof(double))) {
5      printf(``ERROR: vecMult() failed to allocate memory\n'');
6      exit(-1);
7    }
8    for(i=0;i<n;i++) out[i] = x[i]*y[i];
9    return(out);
10 }
```

- This version works ...

vecMult() – again

- ... but it is dangerous – what happens here?

```
1  int n;
2  double *x,*y,*z;
3  /* ... x and y allocated and assigned values ... */
4  z = vecMult3(vecMult3(x,y,n),x,n);
5  x = vecMult3(z,y,n);
```

Correct Usage of vecMult3()

```
1  int n;
2  double *x,*y,*z,*temp;
3  /* ... x and y allocated and assigned values ... */
4  temp = vecMult3(x,y,n);
5  z = vecMult3(temp,x,n);
6  free(temp);
7  free(x);
8  x = vecMult3(z,y,n);
9  free(z);
```

- Clumsy and error prone
- First version of vecMult() is better

  `void vecMult(double *out, double *x, double *y, int n);`

Example – Matrix

- Use struct to create a matrix that knows its size: Recall from lecture 1:

```
1 typdef struct matrix_s {
2     unsigned short *size; /* vector of sizes */
3     double **a; /* pointer to array of contents */
4 } matrix;
```

matrix.h

```
1  #ifndef MATRIX_H /* do not want to include more than once */
2  #define MATRIX_H
3
4  typdef struct matrix_s {
5      int *size; /* vector of sizes */
6      double **a; /* pointer to matrix of contents */
7  } matrix;
8
9  matrix newMatrix(int size[2]);
10 void freeMatrix(matrix *a);
11
12 #endif /* ifndef MATRIX_H */
```

newMatrix()

```
1  matrix newMatrix(int size[]) {
2    matrix a;
3    int i;
4    if (!(a.size = malloc(2*sizeof(int)))) {
5      printf(``ERROR: newMatrix() - failed to allocates a.size\
6      exit(-1);
7    } /* error checking omitted below */
8    memcpy(a.size,size,2*sizeof(int)); /* a.size[i]=size[i] */
9    a.a = malloc(size[0]*sizeof(double*));
10   a.a[0] = malloc(size[1]*size[0]*sizeof(double));
11   for(i=1;i<size[0];i++) {
12     a.a[i] = a.a[i-1]+size[1];
13   }
14   return(a);
15 }
```

freeMatrix()

```
1  void freeMatrix(matrix *a) {
2     free(a->a[0]);
3     free(a->a);
4     free(a->size);
5     a->a = NULL;
6     a->size = NULL;
7  }
```

Usage

```
1  #include "matrix.h"
2
3  int main() {
4    int size[2];
5    int i,j;
6    matrix m;
7    size[0] = 10;
8    size[1] = 2;
9    m = newMatrix(size);
10   for(i=0;i<size[0];i++) {
11     for(j=0;j<size[1];j++) {
12       m.a[i][j] = i*j;
13     }
14   }
15   freeMatrix(&m);
16   return(0);
17 }
```

To compile: `gcc matrix.c main.c`

Using C with other Programs

- APIs – C works with (nearly) everything, but not so easily
- Matlab API – MEX
- Stata – plugins

MEX

- Lets you:
  - Write function in C and use it in Matlab – mexFunction()
  - Access Matlab arrays from C – mxFuncName()
  - Call Matlab from within C – engFuncName()

Writing MEX-files

```
1  #include "mex.h" /* header file for matlab API */
2
3  void mexFunction  /* entry point from matlab */
4  /* Always takes these arguments */
5  (int nlhs,                  /* number of left hand side arguments
6   mxArray *plhs[],           /* pointer to lhs*/
7   int nrhs,                  /* number of rhs arguments*/
8   const mxArray *prhs[]) /* pointer to rhs arguments*/
9  {
10   /* body of function */
11 }
```

- Always begin with this
- mxArray is a type defined in "mex.h"
- Use mx functions to manipulate mxArrays

Working with mxArray

```
1  { /* inside mexFunction() */
2    double *x,*y;
3    int m,n;
4
5    x = mxGetPr(prhs[0]) /* now x points to begining of the arr
6                            argument passed to the function */
7    m=mxGetM(prhs[0]); /* m by n is the size of x */
8    n=mxGetN(prhs[0]);
9
10   plhs[0] = mxCreateDoubleMatrix(m,n,mxREAL);
11   /* creates space in Matlab to hold output */
12
13   function(y,x,m,n); /* some function that operates on x and
14   return;
15 }
```

More Useful Commands

- There are lots – Full list
- mxAlloc() works like malloc() except Matlab manages memory and will automatically free it when you function exits
- mxAssert() – debugging
- mexPrintf() instead of printf ()

fibMexLoop() with Error Checking

```
1  void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
2  {
3    int n;
4     if (mxGetNumberOfElements(prhs[0])!=1) {
5       mexPrintf("Only scalar input allowed\n");
6       plhs[0] = mxCreateDoubleScalar(mxGetNaN());
7       return;
8     }
9     n = mxGetScalar(prhs[0]);
10    if (n<0 || n!=mxGetScalar(prhs[0])) {
11      mexPrinf("Only non-negative integers allowed\n");
12      plhs[0] = mxCreateDoubleScalar(mxGetNaN());
13      return;
14    }
15    plhs[0] = mxCreateDoubleScalar(fibLoop(n));
16 }
```

Stata plugins

- Lets you:
  - Write C functions that can be used in Stata
  - Access Stata variables, macros, and matrices from C
  - Print messages and errors on Stata's screen
- Simpler than Matlab's MEX
- Can only access and modify existing variables and matrices, cannot create new ones

Writing Stata plugins

```
1 #include "stplugin.h" /* header file for Stata API */
2
3 STDLL stata_call /* entry point from Stata */
4 /* Always takes these inputs (same as main for command line p
5 (int argc,      /* number of arguments */
6  char *argv[]) /* string vector containing arguments */
7 {
8   /* Body of function */
9 }
```

Accessing Stata Variables

```
1  { /* inside stata_call() */
2    /* for future compatibility, use datatypes defined in stplu
3    ST_int nObs = SF_nobs(); /* get number of observations */
4    ST_int nVarInData = SF_nvar(); /* number of variables in da
5    ST_int nVarsPassed = SF_nvars(); /* number of variables in
6    ST_double val;
7    /* loop over observations that satisfy ``if'' and ``in'' co
8    for(int j = SF_in1(); j ≤ SF_in2(); j++) {
9      if(SF_ifobs(j)) {
10       /* square 1st variable, store result in 2nd */
11       SF_vdata(1,j,&val);
12       val *= val;
13       SF_vstore(2,j,val);
14       /* would use SF_sdata and SF_sstore for strings */
15     }
16   }
17 }
```

More Commands

- Matrices: SF_mat_el(), SF_mat_store(), SF_col(), SF_row()
- Macros and scalars:
  SF_macro_save(), SF_macro_use(), SF_scal_save(), SF_scal_use()
- Display: SF_display(), SF_error()
- Missings: SF_is_missing(), SV_missval
- That's everything

Stata Fibonacci

```
1  STDLL stata_call(int argc, char *argv[])
2  { ST_int      j;
3    ST_double   val,fib;
4    ST_retcode  rc ;
5
6    if(SF_nvars() != 2) {
7      return(102) ;       /* wrong number of variables specified
8    }
9
10   for(j = SF_in1(); j ≤ SF_in2(); j++) {
11     if(SF_ifobs(j)) {
12       if(rc = SF_vdata(1,j,&val)) return(rc) ;
13       fib = SF_is_missing(val)? SV_missval :
14             (ST_double) fibLoop((unsigned int) val);
15       if(rc = SF_vstore(SF_nvars(), j, fib)) return(rc) ;
16     }
17   }
18   return(0) ;
19 }
```

Debugging

- Debuggers
  - Within IDE
  - gdb – command line, or better yet, within emacs (`<Alt-x> gdb`)
    - ★ GDB Tutorial
  - More debuggers
- `lint` – like Matlab's `mlint`

Memory Related Errors

- Memory bugs – leaks, illegal addressing – are very difficult to diagnose
- Effect of memory bugs can depend on program inputs, compiler options, length of program execution, etc.
- Responsible for many computer viruses
- See Techniques for memory debugging
- Tools:
  - Valgrind – indispensible Intro to Valgrind
  - Electric Fence

Input and Output

- Input
  - FILE* fopen(char name[],char *mode)
  - char* fgets(char *string, int size, FILE *f)
  - int scanf(char *format, ...) also, sscanf, fscanf
  - char *gets(char *s) – use with caution
- Output
  - int printf(char *format, ...) also, sprintf, fprintf
- more, see `man string.h`

Strings

- size_t strlen (const char *)
- char *strchar (const char *s, char c)
- char * strstr (const char *needle, const char *haystack)
- int strcmp(const char *s1, const char *s2)
- char *strtok (const char *s,const char *sep)
- more, see `man string.h`

Reading Files

- If very strict about file format and not careful about errors, can get by with fopen, fscanf , fclose
- More flexible formats and more careful error checking requires more care
- Example: `readcsv.c` – reads a bunch of numbers from a comma separated text file, and then prints them on the screen

Exercises

1. Run a program in a debugger. Figure out how to set breakpoints, move around in the call stack, and display the contents of variables.
2. Modify the Fibonacci program for Matlab so that it works with arrays. Given an array of integers, it should return return an array of the same size with each of the corresponding Fibonacci numbers.
3. Improve readcsv.c in one or more of the following ways:
    1. Make it stores the column or row from which it read each number.
    2. Make it store cells with non-numeric content.
    3. It behaves unexpectly for input such as: “ `1, , 3pm,` ”. Make it do something sensible in these cases.