Midterm #2 Review

1 Time and Location

The midterm will be given in class during normal class hours, 11:00a.m.-12:15p.m., on November 7th.

2 Format

The midterm will be open-book (Kernighan and Ritchie), open-notes, closed everything else. In other words, you can bring any material that you have accumulated in class and recitation, but nothing else. Also, no calculators are allowed during the exam.

3 Scope

The midterm will cover all the material introduced since the last midterm, given on October 1st–essentially covering the material from lecture 11, given on October 3rd, up and including lecture 20, given on November 5th. In other words, the midterm is not cumulative. Here’s an outline of the topics covered in that period. Disclaimer: this is not meant to be an absolutely water-tight complete list of topics. If there is a topic not present in this list, it may still show up on the midterm. However, the list is a pretty good first-cut at what we have covered.

I. String Review
   A. Character utilities
      1. #include <ctype.h>
      2. int isupper(int c)
      3. int islower(int c)
      4. char toupper(int c)
      5. char tolower(int c)
   B. Strings: arrays of characters
      1. Declaration and initialization
         a. "" syntax
         b. Termination by NULL or ’\0’ character
      2. Use in printf/scanf
         a. "%s" formatting placeholder
         b. In scanf, no need for "&" syntax
c. `sprintf`
data. `sscanf`

3. String utilities
a. `#include <string.h>`
b. `int strlen(char *s)`
c. `strcpy(char *s1, char *s2)`
d. `int strcmp(char *s1, char *s2)`

C. Code examples
1. `parse.c`
   a. "parsing" and delimiters
   b. Redirecting standard input and standard output
   c. Dictionary: array of strings or 2-D array of chars
d. Forward declaration (use of "char *", "char []", and "char [][]")

II. Pointer Arrays
A. Arrays of pointers instead of data values
   1. Pointers to pointers
   2. Pointers to arrays
B. Layout in memory
C. Relationship to multi-dimensional arrays
   1. Similarity in referencing / indexing
   2. Difference in layout
   3. Inner dimensions of arrays of pointers to arrays do not have
to be of the same size
   4. Difference in passing as arguments to functions
D. Command-line arguments
   1. Can pass arguments to main
   2. `main (int argc, char *argv[])`
E. Code examples
   1. `args.c`

III. Structures
A. Structure Basics
   1. Declaration of structure template
      a. Use of `typedef`
   2. Declaration of variables
      a. Combining variable and template declaration
      b. Initializing structs
   3. Accessing structure members
      a. "." syntax
      b. Setting all fields of a structure to the corresponding
fields of another structure in one statement
   4. Pointers to structures
      a. The address of a structure is the address of the first
element in the structure
      b. "->" syntax for derefencing a pointer to a struct
   5. Arrays of structures
      a. Pointer arithmetic
   6. Structures as function parameters
      a. Passing structs call by value
      b. Passing structs call by reference
c. Returning structs from functions

B. Code examples
1. structure.c
   a. sizeof()
2. array_struct.c
   a. Sanity check command-line arguments
   b. Parsing from data file using fscanf
   c. Casting

IV. Dynamic Memory Allocation
A. malloc
1. Statically allocated vs dynamically allocated variables
2. 3 different data memory regions in C
   a. stack
   b. static data region
   c. heap
3. void *malloc(int)
   a. allocates ‘int’ contiguous bytes on the heap
   b. returns a generic pointer, void *, to the first byte
   c. casting of generic pointers to specific pointer types
   d. returns NULL on failure
   e. #include <stdlib.h>
4. malloc’d objects have no symbolic name; access only through pointers

B. Related functions
1. void free(void *)
   a. Deallocates memory allocated by malloc
   b. Prevents memory leaks
2. void *calloc(int, int)
   a. Similar to malloc
   b. Separates ‘sizeof’ and ‘number of’ elements into different parameters
3. void *realloc(void *, int)
   a. Takes object pointed to by ‘void *’, and changes its size to ‘int’

C. Code examples
1. malloc.c
2. matadd.c
   a. Dynamic allocation of pointer arrays

V. Linked Lists
A. Basics
1. Self-referencing structures
   a. Declaration of ‘link nodes’
   b. Has a field, usually called ‘next’, that points to a struct of the same type
2. Data structure layout
   a. Multiple link nodes are chained together through ‘next’ pointers
   b. Components: head pointer, link node, tail node with NULL ‘next’
3. Benefits
a. Number of objects in the list can vary dynamically
b. Insertion/deletion to/from arbitrary point in list is easy

B. Operations
1. Traversal
   a. ‘‘p = p->next’’
2. Insert
   a. Head insertion
   b. Middle insertion
   c. Tail insertion
3. Delete
   a. Head deletion
   b. Middle deletion: must find ‘‘previous’’ node

C. Doubly-linked lists
1. Data structure layout (add a ‘‘prev’’ pointer to every self-referencing structure, and a ‘‘tail’’ pointer)
2. Traversal -- same as singly-linked list
3. Insert / delete
   a. Use ‘‘prev’’ pointer to access previous node’s ‘‘next’’ pointer
   b. Only requires 1 pointer to search for insert/delete point, and to perform insertion/deletion

D. Code examples
1. list.c
2. students-array2.c
3. students-list.c
4. students-double.c

VI. Advanced Linked Data Structures
A. Hash tables
1. Elements
   a. Key
   b. Hash function
   c. Bucket
2. Implementation
   a. Pointer array
   b. Given key, hash function maps to pointer array index or bucket

B. Trees
1. Data structure layout
   a. Root pointer
   b. Each tree node has 0 or more children
   c. Root node has 0 parents, leaf nodes have 0 children
   d. Depth
   e. Height
2. Applications
   a. Storing hierarchical data
   b. Sorting
3. Ex: Binary search tree
   a. Tree with node fanout = 2
   b. Nodes contain data, each with unique key
   c. Left sub-tree of a node contains only nodes with keys <
node’s key.

d. Right sub-tree of a node contains only nodes with keys >
   node’s key.

e. Both left and right sub-trees must also be BSTs

C. Code examples
1. list-search.c
2. hash-search.c
3. tree-search.c

VII. Recursion
A. Principle of Induction
1. Inductive vs iterative formulation of problems
   a. Inductive: break solution to problem into a smaller
      problem of same type (and some primitive operations)
   b. Math formulas formulated in this way are called
      recurrences
   c. Base case and inductive case
2. Implement inductive formulation as a recursive function
3. Forward repetitions, followed by backward repetitions during
   ‘return’ from inductions

B. Classic recursive problems
1. Traveling salesman problem
   a. given N cities and cost for traveling from city i to j,
      find tour that visits all cities once with minimal cost
   b. (N-1)! possible tours
   c. Solve recursively by expanding partial tours
2. Tower of Hanoi
   a. Move N disks from source peg to destination peg, using
      middle peg as temporary space
   b. Solve recursively by moving N-1 disks from source to
      middle peg, Nth disk from source to destination peg, and
      N-1 disks from middle to destination peg

C. Code examples
1. tree-search2.c
2. tsp.c
3. hanoi.c