1 Time and Location

The midterm will be given in class during normal class hours, 11:00a.m.-12:15p.m., on Tuesday October 3rd.

2 Format

The midterm will be open-book (Kernighan and Ritchie, and Loukides and Oram), 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 or laptops are allowed during the exam.

3 Scope

The midterm will cover all material from the first day of class up and including lecture 10, given on Thursday September 28th. Here’s an outline of the topics covered thus far. Disclaimer: this is not meant to be an absolutely water-tight complete list of topics. In other words, if there is a topic not present in this list, it may still show up on the midterm. However, it is a pretty good first-cut at what we have covered.

I. Introduction
   A. Declarative vs Imperative knowledge
      1. Declarative: "What is true" knowledge
      2. Imperative: "How to" knowledge
      3. Programming: act of expressing "how to" knowledge
   B. Program complexity
      1. Metric: Source Lines of Code
      2. Complexity management techniques
         a. Divide and conquer
         b. Abstraction
         c. Good coding conventions
   C. C language features
      1. General purpose
      2. Few features
      3. High performance
      4. Closest to machine’s low-level representation
II. ENEE 140 Review
(See handout #2)

III. Functional Decomposition
A. Motivation
1. Separating use from implementation
   a. Divide and conquer
   b. Improves code understanding
   c. Easier to debug
   d. Facilitates incremental project implementation
2. Code reuse
3. Code management
   a. Can change implementation without affecting its use
B. How to decompose
1. Art, not science
2. General rules
   a. Logical units
   b. Code reuse
   c. Hierarchical decomposition
C. Top-down design
1. Identify modules starting from top-level to bottom-level
2. When coding, apply "wishful thinking"
3. Code factoring (bottom-up)
D. Multiple modules
1. Create projects via multiple .c files
2. Compiling
3. Header .h files
4. Scoping global variables, use of "extern"
E. Code examples
1. sheet.c
2. chess.c (not provided)
3. sqrt3.c, sqrt3.h, support3.c

IV. Testing
A. Test vectors
1. Set of inputs
2. Set of expected outputs
B. How to run
1. "<' redirects stdin from a file
2. "'>", ">!" redirects stdout to a file
3. "diff" compares two files
C. How to write
1. Coverage
2. Test suite
D. Unit/module test vectors
1. Motivation
   a. Supports functional decomposition discipline
   b. Enables testing during incremental project build
   c. Allows debugging fewer bugs at a time
   d. Higher coverage of inputs possible
2. Same inputs/expected outputs + driver code
3. Removing main()  
   a. Place main() in a separate .c file  
   b. Use #ifdef or #ifndef  
E. Incremental build and test  
   1. Design functional decomposition  
   2. Implement a function  
   3. Write test vectors for new function  
   4. Run test and debug  
   5. Re-run test suite  
   6. Goto 2 (implement another function)  
F. Shell scripts  
   1. Shell interpreter  
   2. #!/bin/csh  
   3. echo  
   4. foreach  
   5. variables: $ syntax  
   6. chmod  
G. Code examples  
   1. search.c  

V. Debugging  
A. Overview  
   1. Types of bugs  
      a. Compile-time  
      b. Link-time  
      c. Runtime  
   2. Debugging approaches  
      a. Code inspection  
      b. printf  
      c. Debuggers  
   3. Instrument/compile/run/debug loop  
B. GDB  
   1. "-g" compilation flag  
   2. Running programs in gdb  
   3. Printing variable values  
   4. Breakpoints  
   5. Stepping  
C. Code examples  
   1. students-array.c  

VI. Pointers  
A. Number systems  
   1. Binary encoding of numeric values  
   2. Converting from binary to decimal, and decimal to binary  
   3. Base-n encoding  
   4. Converting from base-n to decimal, and decimal to base-n  
   5. Hexadecimal and octal representations  
   6. C syntax for binary, hex, and octal  
B. Programmer's view of memory  
   1. To programmer, memory is a contiguous set of locations, each  
      containing 1 byte of data  
      a. Locations contain code, stack, and global variables
b. In a 32-bit machine, there are $2^{32}$ locations
c. Notion of an ADDRESS

2. Word-wide view of memory (instead of byte-wide view)
3. Layout of different datatypes in memory
4. Address of variables in memory (ints, shorts, floats, doubles, arrays, etc.)

C. Pointer Basics
1. A pointer is a variable that stores the address of another variable
   a. Provides another way to refer to a variable.
   ==> A variable’s address is another "name" for the variable--i.e., a runtime name
   b. Can perform computation on address names
2. Declaration
   a. "*" syntax
   b. Declare the name of the pointer, and the type it points to
3. Setting pointers
   a. 
4. Access through pointers
   a. "*" syntax
   b. Indirection, or dereferencing a pointer
5. Printing pointers
   a. 
   b. Pointers are usually printed in hex representation

D. Pointer arithmetic
1. Adding a value to a pointer changes the pointer in increments of the size of the element being pointed to
2. Operations
   a. Adding a value to a pointer
   b. Subtracting a value from a pointer
   c. pointer "++", pointer "--"
   d. Subtracting 2 pointers
3. Only makes sense when pointer points into an array
4. Equivalence of "*(a + i)" and "a[i]"
5. Equivalence of pointers and arrays
6. Differences between pointers and arrays

E. Pointer and Function Arguments
1. Call by value
   a. Formal parameters are copies of actual parameters
   b. Modification of formal parameter values have no side effects
2. Call by reference
   a. Formal parameters are pointers to variables in actual parameters list
   b. Modifications through formal parameter pointers have side effects
   c. Relationship to return values
3. Arrays are always passed call by reference

F. Code examples
1. layout.c
   a. "-c" flag during compilation
b. od -A x -X layout.o
2. pointer.c
3. pointer-arith.c
   a. sizeof()
   b. traversing an array using pointers