This list is intended to help you study for the material that may appear on the midterm exam. Do NOT use it as an exhaustive list of what you need to know. On the exam, we will provide a reference sheet or hints as needed: e.g., shell command arguments/options, shell variables, other syntax.

I. Shell basics
   A. echo
   B. users
      1. file read/write/execute permissions
      2. user home directory
   C. exploring and manipulating files and directories
      1. filesystem basics
      2. cd, ls, mkdir, cat, less, rm, mv, cp
      3. filename expansion (globbing)
   D. processes
      1. ps, to list process information
      2. top, to monitor running processes
      3. kill, to send signals to running processes
      4. suspend, run in background, run in foreground (cmd &, fg, bg)
      5. exit code
   E. other shell commands
      1. aliases
      2. history
   F. I/O redirection (stdout, stdin, stderr, append/overwrite)
   G. command composition
      1. pipes: x | y sends x’s stdout to y’s stdin
      2. x && y: run x then run y only if x is successful
      3. x || y: run x then run y only if x isn’t successful
   H. shell variables
      1. command line arguments variables
      2. user variables
      3. exit code variable
      4. shift for arguments processing
      5. export
      6. quoting/strings
         a) all variables are strings
         b) “ “ to use spaces, allows variable substitutions
         c) ’ ’ suppresses all substitutions (for literal string)
   I. shell scripting
      1. control
         a) if and test
         b) for v in w1 w2 … wn
      2. basic shell arithmetic
   II. text processing
      A. regular expressions
         1. a|b alternation
         2. a* repetition
         3. ab concatenation
      B. grep
1. additional constructs that are there for convenience
   a) \[a-z\] is one character from the range
   b) p+ is pp*
   c) \[^abc\] is one character that is NOT a,b, or c
   d) . matches any
   e) $ matches end of line
   f) ^ matches beginning of line
2. -o for returning just the match
3. * is greedy, so often need to use \[^\...]\* to avoid matching too far
4. familiarity with egrep or grep regular expression syntax
5. groups and back references
   a) a group goes in ( )
   b) reference the first group match with \1, second with \2, …
   c) example: find palindromes from a list of words
C. sed
   1. using sed 's/…/…/' for basic find replace line-by-line
   2. add g to the end for all matches on each line, without g is just first match on each line
   3. sed -n 's/…/…/p': print only lines with a match
   4. use back references to replace text;
      a) e.g., sed 's/([a-z])/\1\1/g' doubles all lowercase letters
   5. addresses
      a) sed '<address> <command>'
      b) apply find/replace only on line 54, sed '54 s/SAMPSON/RINGO/g'
      c) apply delete on lines that match the pattern, sed '/SAMPSON/ d'

III. C programming
A. if/else, for, break
B. function definitions and function calls
   1. everything is pass-by-value (except arrays)
C. declarations and definitions
   1. declaring a function by writing just its prototype
   2. defining a function (exactly once) by writing an implementation
   3. declaring/defining variables
      a) int x; // allocates memory for variable x; does not set its value
      b) just be aware that the above example is a definition of x. To only declare x uses extern
D. storage, scope, lifetime
   1. kinds of variables (local, global, static global, static local)
   2. identify where in memory different kinds of variables are stored
   3. identify the scope of different kinds of variables (where in the code it is visible)
   4. identify the lifetime of different kinds of variables (when during execution the memory is valid, i.e. when is it allocated, when is it deallocated)
   5. types: int, char, pointers, arrays
E. Memory
   1. code, stack, heap, global data, global constants
      a) what data does in each one
      b) is each one executable, readable, writeable
   2. The stack
      a) each function call gets a frame where return address and local variables (including arguments) are stored
b) frame is allocated on the call and deallocated on return

F. C preprocessor
1. transforms the code files before they are given to the compiler
2. `#include “foo.h”` include a file here (typically to get declarations)
3. `#define PI 3.14` symbolic constants
   a) textual replacement, NOT a variable

G. Pointers
1. allocate a pointer
   a) `int* x;`
2. setup a pointee
   a) `int j = 10;`
3. setup the pointer
   a) `x = &j;`
4. dereferencing
   a) `*x = 5; *(x+0) = 5; x[0] = 5`
   b) `int y = *x; int y = *(x+0); int y = x[0];`
   c) `*(x+2) = 6; x[2] = 6;`
5. address-of (&)
6. identify when dereferencing a pointer is a bug
   a) it could crash (segmentation fault or seg fault)
   b) but might not crash, in which case the program reads or writes some memory you didn’t intend to (typically much worse than a crash)
7. identify dangling pointers
8. pointer arithmetic. Addition increments by elements, not by bytes
   a) `T* x;`
   b) `T* h = x+i; // h holds byte address: x + sizeof(T)*i`
9. pointers to pointers; argv is the most familiar example
10. using pointers to pass data in and out of a function call by reference

H. Arrays
1. an array variable is interpreted as a pointer when used in an expression, including a function call
   a) therefore, arrays are passed by reference
2. declaring an array in a function body allocates it on the stack
   a) `char x[10]; // allocate 10 chars contiguously on the stack`
3. char arrays
   a) a string is a char array where we encode the length by ending with null terminator character ‘\0’. Need space to store it, too!
   b) what does the memory look like for
   (1) `char g[7] = “BATMAN”;`
4. argc and argv

I. Left values and right values (lvalues and rvalues)
1. a left value has a location in memory
2. a right value is just a concrete value; doesn’t have a location in memory
   a) values are numbers or addresses
3. understand what is allowed as the left of an assignment
4. `*` converts an rvalue (specifically, an address) into an lvalue (the memory location pointed to)
5. `&` converts an lvalue (anything with a memory location) into an rvalue (an address)

J. IO
CSE 374 Midterm exam concepts

1. printf
   a) replace %d (int), %p (pointer) in format string with a value
2. scanf
   a) formatted input