Topics for CSE 374 Winter 2014 Final (Tentative)

Note: The abbreviation "**HYCSBWK**" (and variations) in some of the slides was first created by Grossman and Perkins, and stands for "hope you crash soon, but who knows?". It is meant to indicate that doing something potentially "bad" in C may not cause your program to crash right away, unlike some other languages such as Java that perform run-time checks that may catch such problems sooner and halt, possibly with more graceful error messages.

Final Topics table of contents:

- C programming language
- Pre-processor instructions
- Debugging (GDB)
- Memory
- Structs, Linked Lists, Trees
- Version control (SVN)
- Makefiles

Homeworks to focus on:

- HW4: processing strings, initial use of `malloc`, examining `argc` and `argv` (parameters to `main`)
- HW5: trie (`struct`, tree and/or linked list, `malloc`, `free`); initial use of `gdb` and `make`
- HW6: memory management (`struct`, pointers, linked lists, arrays allocated on the heap, pointer arithmetic, casting, `malloc` and `free`); initial use of `svn`

Not On Final:

- 18-linking
- C++ (19-cpp-intro; 20-cpp-subclass; 21-funcptrs)

Final Topic List:

- **C programming language**
  - Some similarities to Java control structures and syntax (Java came later)
  - Much more "unsafe"; much more for the programmer to keep track of explicitly (checking for falling off the end of an array – no length primitive, dynamic memory management, etc.)
  - Procedural instead of object-oriented; no classes
  - Control constructs:
    - `while`, `if`, `for`, `switch`, `break` (`case`s without a `break` in `switch` "fall through"); use `int` for Boolean (0 or `NULL` is *false*, anything else is *true*)
  - Functions:
    - Function prototype: `int twice(int x);`
    - Function definition: `int twice(int x) { return 2*x; }`
  - `printf` and formatting arguments (`%s`, `%d`, `%f`, `%p`, etc.)
  - Study Resources:
    - Lecture slides: 07-c-intro, 08-c-misc
    - Homework: HW4, HW5, HW6
    - Programs: `hello.c`, `printargs.c`, `magic.c` (annotated versions also available)

- **Concept: Pre-processor instructions**
  - Including references to library functions or user-created header files

```
#include <stdio.h>
#include "myHeader.h"
```

  - Defining constants

```
#define THE_ANSWER_TO_EVERYTHING 42
#define NORMAL_BODY_TEMP_F 98.6
#define GREETING "Wassup, Dawg?"
```

- Including contents of header files (`*.h`) only once during linking

```
#ifndef MYHEADER_H
#define MYHEADER_H
. . .
#endif
```

- Study Resources:
  - Lecture slides: 8-c-misc; 13-cpp
  - Homework: HW4, HW5, HW6

- **Concept: Debugging**
  - Debugging tools
    - Many modern integrated development environments (IDEs; for example, Eclipse) have debuggers for various languages, and they may also have graphical user interfaces (GUIs). The debugging tool we looked at in class, `gdb`, is command-line driven (text-based) rather than having a GUI. The types of things `gdb` can do, however, are similar to those you may have seen in a GUI-style debugging tool.
  - What can we use a debugging tool for?
    - Examine control structure of a program (which instructions get executed in which order; does this match how we thought it was working?)
    - Examine values at various points of execution (do these values match what we expected them to be for what we intended with our code, did we forget to initialize a value, is our arithmetic expression evaluating how we expected, etc.?)
    - Systematic rather than random use (I ***think*** the code is doing X, then test hypothesis by using debugger; your own brain is the best tool you have, use other tools to confirm or deny your mental model of what your program does)
    - Inspect reasons for a hard crash (Run until the crash, then determine what "signal" was given (segmentation fault, abort, etc.); also find out at which line the crash occurred (real problem could have happened earlier in execution, so work backwards from that point))
  - Usage:
    - Compile with –g flag for `gcc` to get useful debugging info such as "symbols" (names of variables and functions); otherwise will not be able to do much
    - `$ gdb` <*filename of executable*> <*any command-line arguments for executable*>
  - Commands
    - Run, step, next, continue, finish (execute the instructions of your program)
    - Breakpoint (pause execution at a specific line or entry to a specific function)
    - Print expression, display expression, info args, info locals (examine values at various points of execution)
      - Probably one of the more useful strategies in addition to extra print statements in code during development
    - List (list the source code of the current focus of execution)
    - Backtrace (trace the stack of function calls that led to the current focus of execution)
    - Frame, up, down (look at the stack (activation record) for the current function; move focus up or down the call stack)
  - Study Resources
    - Lecture slides: 11-gdb
    - GDB manual: http://www.gnu.org/software/gdb/documentation/
    - Homework: HW4, HW5, HW6
    - Links on calendar at 01/31, 02/03: `gdb reverse[1-4] demo`; `reverse[1-4].c`

- **Concept: Memory**
  - Address space of a running process provided by the operating system (OS)
  - Both data and code are in this space
    - Attempts to access currently unused portions of this space causes a "segmentation fault" error

    - Memory management of "the stack" is automatically handled (declarations of variables, activation records of function calls)
    - Memory management of "the heap" must be handled explicitly by the programmer (`malloc` and `free`)
- Declare a pointer to a type
    - `int * y; int* y; int y; int*y; // matter of style`
    - Allocate space for a **pointer** on the stack (not space for the type of thing pointed at)
- Dereference a pointer (what is the pointer pointing at?)
    - `*y`
    - an expression, compute a value
- `malloc`
    - Arrays whose length depends on runtime information are allocated on the heap
    - `t *myArr = (t*) malloc(e * sizeof(t));`
    - returns a pointer to `void` which is cast to a pointer to `t` by `(t *)`
    - still need to initialize (use a loop!); otherwise garbage values
    - `myArr[0]`, `myArr[e-1]` are first and last elements
    - `myArr[5]` and `myArr+5` are the same location
- `calloc` takes `e` and `sizeof(t)` as separate arguments and initializes all bytes to 0
- `free`
    - must explicitly `free` memory gotten through `malloc`, `calloc`, etc.
    - no garbage collection in C
    - memory leak (lost access to an area in memory before freeing it)
    - dangling pointer (attempt to use a pointer address after it has been freed, or `free` it twice)
    - normally memory released by operating system after program terminates
    - good practice to have program "clean up" its own heap memory usage
- Study Resources:
    - Lecture slides: 7-c-intro, 8-c-misc, 9-c-values, 10-c-heap
    - Programs: `lec9_slide10_A.c`, `lec9_slide10_B.c`, `lec10-slide7.c`
    - Homework: HW4, HW5, HW6
    - References: http://cslibrary.stanford.edu/102/PointersAndMemory.pdf, http://cslibrary.stanford.edu/104/ (Binky pointer video), http://cslibrary.stanford.edu/106/ (pointer basics, companion text to Binky video)

- **Structs, Linked Lists, Trees**
    - Left-expressions evaluate to locations
    - Right-expressions evaluate to values
    - `x.f` for field access if `x` is not a pointer
    - `x->f` for field access if `x` is a pointer; is the same as `(*x).f`
    - `struct` arguments to functions are copied (call-by-value)
        - if a large structure, use a pointer to the `struct` instead (call-by-reference)
        - (side note, recall that array arguments are always promoted to pointers, so they are call-by-reference also)
    - Casting
        - Can't cast one `struct` to a different one (could be different sizes in memory)
        - Can cast one pointer type to another (ex, cast a void pointer to a pointer to a struct)
        - Can be "unsafe"; C does not do any further checking (Java is more "strongly typed")
            - May only get a warning (or not) when attempting to put a double into an int in C (will lose information) without casting; Java must explicitly cast
    - Dynamically allocate linked list or tree nodes with `malloc`
    - Study Resources:
        - Lecture slides: 9-c-values (for review of left vs. right expressions (or lvalues, rvalues)), 12-c-structs
        - Programs: `struct.c, list.c`
        - Homework: HW5, HW6
        - Other References: CS Stanford http://cslibrary.stanford.edu/103/

- **Concept: Version Control**
  - Version control tools
    - The version control tool we used in class was called subversion (`svn`). There are others (`cvs`, `git`, etc.) that vary in how they work "under the hood", but some more or less common ideas across them is that you have a shared, possibly remote, "repository" holding the latest and greatest code that has been "committed" from everyone, and each developer has a "local working copy" to develop against.
  - What would we use version control for?
    - In class, the idea behind using `svn` on HW6 is that you and your teammates could share a single repository for "committed" code. Each person "checks out" a "local working copy" in which to perform development, and "commits" his/her changes to the common remote repository. Developers should call the "update" command prior to committing code to make sure any changes to the code in the repository (from their teammates) gets merged into their own working copy first. This management of merging changes and detecting if files conflict is one of the main advantages of a version control tool. Sharing files manually by sending them around in email or uploading files to a common "drop box" is very error prone because it would be easy to accidentally overwrite someone else's changes.
    - BTW, you can even use version control when you are the only developer on a project. Version control tools usually have some feature that can "tag" a collection of files at a particular revision so that you can "go back" to that revision later (handy if development really went down a wrong path and you want to start over from a known working point). We didn't show the "tag" idea in the notes so it won't be on the test, but it's good to know about. If you plan to do any significant programming as part of your job, whether or not you are on a team, you should get in the habit of using some kind of version control tool.
  - Commands:
    - Import
    - Update
    - Commit
    - List
    - Add
    - Copy
    - Delete
  - Study Resources:
    - Lecture slides: 16-svn
    - SVN manual: http://svnbook.red-bean.com/
    - SVN command demos linked on calendar at 02/24: (wi13 shell command history) `natasha`, `boris`; (wi14) step-by-step movie showing on-screen results of commands
    - Homework: HW6

- **Concept: Makefiles**
  - A "helper" program to automate what gets (re)compiled and how (other programs besides `make` exist; for example, `ant`, `maven`, "projects" in IDEs, etc.)
    - Scripts for executing commands ("rule" = target, source(s), command(s))
    - Dependency information to avoid unnecessary work (`gcc -M` to help you get started)
    - http://xkcd.com/303/ (BIG projects can take a LONG time to recompile EVERYTHING, so the idea behind `make`, etc., is to only recompile what is necessary, such as source files that have changed)
  - Study Resources:
    - Lecture slides: 14-make
    - Program files: `talk.tar`, `talk_DAG.pdf`, `revised_struct.tar`
    - Homework: HW5, HW6
    - Make manual: http://www.gnu.org/software/make/manual/make.html
    - CS Stanford: http://cslibrary.stanford.edu/107/UnixProgrammingTools.pdf