README

After writing this, we realized this document is more of a **condensed version of the lecture notes** than it is a study guide. This document includes *a lot of information* and is likely (at least a little bit) overwhelming. Since Dr. Eggert's exams tend to ask really in-depth and open-ended questions, we didn't feel a study guide that simply listed out topics would be very helpful.

That being said, it's okay to not study everything below! In fact, please take care of yourself and **don't study everything**. As this study guide is designed for the sake of completeness, previous exams will likely be a much better indicator of the content that will appear on the final. Instead, use this document to identify what you might need to know about a particular topic(s). Focus on the topics that confuse you, or even better, focus on the topics that interest you.

On that same note, you also are not expected to know everything about every bullet below. Since the test is open-Google, make sure you, above all, have the conceptual knowledge to know how to *find* the answer to a question. For example, you may forget how to write an if-statement in a shell script. But if you have a solid conceptual grasp of how if-statements in the shell work, doing so is one quick Google search away.

Lastly, if something confuses you, ask questions! Piazza is not only a great place to get help on your assignments, but also a great place to ask conceptual questions. Best of luck on the exam!

– Fall 2020 Learning Assistants

CS 97 Final Study Guide

Questions that may help/challenge your understanding of the material are highlighted in yellow. Note that this study guide only includes material from the lectures. Make sure you are familiar with the assignments as well!

Emacs and Shell

Corresponding lectures: 10/6, 10/8

- Why Emacs?
  - Scriptability (i.e., Emacs Lisp)
  - Introspection
- Emacs commands. You don't have to memorize these – just make sure you're comfortable with how commands in Emacs work.
- Shell and Emacs are both examples of a REPL (read–eval–print loop).
- Shell commands. Again, you don't have to memorize these, but make sure you know how to read documentation!
  - `echo`, `ls`, `rm`, `cat`, `head`, `true`, `false`, `ln`, `ln -s`, and many more!
  - `man` pages (online), Google
- Make sure you can read and write simple shell scripts.
  - Control flow (i.e., if statements, loops), variables, pipes, redirection, etc.
  - Built-in variables: `$1`, `$2`, `$3`, `$?`
- What about Emacs makes it introspective?
- In what ways is Emacs a REPL?
- In what sense does the shell "glue" smaller programs together? Which parts of the shell language make it especially suitable for this purpose (compared to something like Python)?

Aside: POSIX

Corresponding lectures: 10/13

- Portable software: software that works across multiple platforms
- POSIX (Portable Operating System Interface) specifies many commonly used Unix-like functionality
  - OSes that implement POSIX: Linux, macOS, Solaris, Windows (partially), etc.
  - C Functions
  - The Shell (implemented by Bash)
- Why is software portability important? Why are some of the challenges in writing portable software, and what are some ways to address these challenges?
- Why does POSIX have a specification for C functions?

"Little Languages" and Scripting Languages

Corresponding lectures: 10/13, 10/15

- What to consider when choosing a programming language

    - Performance
    - Reliability
    - Ease of use
    - Scaling issues
- Static vs. dynamic type checking
- Approaches to constructing software
    - Using a single language to write your entire application (cf. CS 31)
    - Using many "little languages" that each have unique and focused functionality (e.g., `sh`, `grep`, `sed`, `awk`)
- Regular expressions and `grep`
    - BRE vs. ERE
    - A RegEx cheat sheet
- Meta-execution in scripting languages (when the program "writes" part of itself)
    - e.g. `$( )` and `eval` in shell scripting
- Lisp (LISt Processing)
    - Data types in Lisp. The important ones are integers, floats, strings, cons (i.e., lists), and symbols.
    - Functions and variables
    - Quoting
    - Working with lists (i.e. `cons`, `car`, `cdr`, `nil`)
    - Be comfortable working with Emacs Lisp-specific functions (i.e. `message`, `current-buffer`, `point-max`, `point-min`). Again, you don't have to know what these specifically do, but just make sure you're able to find and read their documentation!
- "Why have three languages [shell, Lisp, and Python]? Why not just one? Why not just use C++?" (quoted from lecture)
- See Economies of scale. How does this relate to software? (Hint: see the lecture notes)
- Lisp is short for LISt Processing. What does Lisp have to do with lists?

Python

Corresponding lectures: 10/20, 10/22, 10/27, 10/29

- CPython vs. other Python implementations such as Jython (runs on JVM) and PyPy
    - CPython "compiles" Python into `.pyc` files, which include Python bytecode that is portable across different platforms
- Everything in Python is an *object*, which consists of a unique identity, a type, and a value.
    - e.g., the string `"eggert"` may have id `4369821040`, type `str`, and value `"eggert"`.
    - Mutable vs. immutable objects
- Python data types
    - Numbers: integers, floats, complex numbers, booleans
    - Immutable sequences: strings, tuples, ranges
    - Mutable sequences: lists, buffers

    - Mappings: dictionaries
    - Callables: functions, classes, methods
- Make sure you can read and write Python code.
    - Functions, classes, subclasses, control flow, namespaces, etc.
    - Refer to the 10/20 and 10/22 lectures for details on sequences/lists, functions, and classes.
- Python modules
    - Modules are typically single Python files (e.g., `foo.py`).
    - Know what happens when you import a module.
    - PYTHONPATH
    - Packages and module hierarchies (e.g., `import cs97.final.solutions`)
    - pip (and why it's really useful)
- Creating your own Python packages
    - When someone installs your package, the `setup.py` file will install all required *dependencies*.
- In languages such as C++ and Java, types such as characters and integers are known as "primitive data types" and are not objects. In Python, *everything* is an object. What are the advantages and disadvantages of Python's approach?
- Why might immutable data structures be easier (and sometimes harder) to work with?
- What are the similarities and differences between C header/source files and Python modules?

Aside: Dependencies

Corresponding lectures: 10/29

- When one part of your software relies on another part, we call it a dependency.
- Build-time dependencies in GNU Make
    - Know how to read and write basic Makefiles.
    - Know how Make avoids recompiling files that are up-to-date.
    - Know why Make is more useful than a simple shell script.
- Installation-time dependencies in pip
    - Understand the concept of dependency graphs and DAGs.
    - Know how pip resolves dependencies using a dependency graph.
- See the Fall 2020 midterm solutions for detailed descriptions/examples of development, build-time, and installation dependencies.
- Instead of using the file's time-stamp, how might Make use checksums to determine if it should recompile a file? (Hint: see lecture notes)
- What restrictions might a tree-based dependency graph have that a DAG-based dependency graph doesn't? Does a tree-based dependency graph even make sense?
- Given what you know about pip, in what way might npm (Node Package Manager) need to deal with dependencies?
- How does semantic versioning help when dealing with installation dependencies? (Hint: see lecture notes)

Networking

Corresponding lectures: 10/29, 11/3

- An overview of networking (don't need to read this, but you may find it helpful)
- In a client-server model, there are two parts
    - The server is the source of information or application state.
    - The clients are peripheral pieces of the application. They request resources from the servers, then display these resources to the user.
    - These are (mostly) synonymous with the terms "frontend" and "backend."
- Other models: peer-to-peer (e.g., BitTorrent), primary/secondary (see lecture notes)
- Performance issues: throughput and latency
- Correctness issues:
    - Out-of-order execution, which may arise when the server and client execute in parallel (addressed with *serialization*, a.k.a. *synchronization*)
    - Out-of-date caches, which may arise when the client caches server state to avoid redundant requests (addressed with *cache validation*)
- Circuit switching (traditional telephones) vs. packet switching (the Internet)
    - Concise explanation of the differences
    - Know why circuit switching guarantees throughput and latency, while packet switching does not.
- Network layers (useful diagram)
    - Link layer involves communication between *hardware* (network cards)
    - Internet layer involves communication between *IP addresses* (packets)
    - Transport layer involves communication between *hosts*, a.k.a. computers (TCP and UDP)
    - Application layer involves communication between *processes*, a.k.a. running programs (HTTP)
- (internet layer) Internet Protocol (IP), which specifies the format of individual packets
- (transport layer) Transmission Control Protocol (TCP) vs. User Datagram Protocol (UDP)
    - Know why TCP is more reliable than UDP.
- (application layer) Hypertext Transfer Protocol (HTTP), which is built atop TCP
    - HTTP vs. HTTPS vs. HTTP/2 (know how HTTP/2 improves on HTTP)
    - Another application layer protocol is Real-time Transfer Protocol (RTP), which is built atop UDP.
- World Wide Web
- How does React fit into a client-server model? (Hint: see the Fall 2020 midterm)
- If you're trying to make a large multiplayer video game, in what ways is throughput important? In what ways is latency important?
- Compare and contrast the advantages and disadvantages of a circuit-switched vs. packet-switched network. Why isn't the Internet built on a circuit-switched network?
- Zoom uses RTP for video calls. Why not HTTP?

HTML, CSS, and JavaScript

Corresponding lectures: 11/3, 11/10

- Know in what way HTML is a tree.
- Know how to read and write HTML.
- Document type definition (DTD)
    - We mostly gave up on DTDs for HTML (they're still used by XML though!), so HTML is now a "living standard."
- Document Object Model (DOM) and how it relates to JavaScript
- Know what CSS is for.
- JavaScript is a scripting language that can be used to manipulate the DOM
    - React and JSX (extension of JavaScript)
    - Know how to read and write JavaScript and React code.
    - JavaScript Object Notation (JSON)
- Node.js
    - Callbacks and event handlers
    - The event loop
    - Node Package Manager (npm)
- Would it be possible to use JSON to model the DOM (instead of HTML)?
- In what way is JavaScript able to achieve "parallelism" with the event loop, despite only running in a single thread?
- JavaScript was never designed to be a server-side programming language, so why do you think Node.js became so popular?

Basic Git

Corresponding lectures: 11/10, 11/12

- Git stores information about a repository in two places:
    - Object database, which contains historical information (i.e., commits)
    - Index file (a.k.a. the staging area), which contains future plans (i.e., creating commits, handling merges, etc.)
- Git commands. Like all other commands, don't feel pressured to memorize these (beyond the basic ones)!
    - `init`, `clone`, `add`, `commit`, `push`, `pull`, `log`, `reset`, `checkout`, `branch`, etc.
- "You're not just writing a program... You're writing a set of *changes* to a program" (Eggert).
- Git branches
    - Know how branch merging works.
- What would be the disadvantages of Git not having an index? Are there any potential advantages?
- As you probably noticed while working on the project, it's not always easy to avoid merge conflicts while working in a team. What are some practices (perhaps involving branches) that you might want to consider adopting to *minimize* the frequency of merge conflicts?

Low-level Programming

Corresponding lectures: 11/12, 11/17, 11/19, 11/24, 12/1

- C and C++ provide advanced features with access to the lowest level of a system
  - Hardware features (like RDRAND), system calls (write(), read())
  - More performant (but also more error-prone) than languages like Python, JavaScript
- C++ ≈ C + extras
  - C++ has classes, templates, exception handling, overloading…
- Stages of C compilation
  - Preprocessing: Resolves `#includes`, `#defines`, `#ifdefs`, etc.
    - You can see the result of this step through `gcc -E`
  - Compilation: Takes a preprocessed C file and outputs a .s assembly file
    - You can see the result of this step through `gcc -S`
  - Assembly: Takes a .s assembly file and outputs a .o object file ("object code")
    - You can see the result of this step through `gcc -c`
  - Linking: Takes one or more .o object files and outputs an executable (e.g., a.out)
- To use a C library, you usually need to do two steps:
  - Add `#include` to satisfy the preprocessing and compilation steps
  - Add `-l<library>` compiler flag to satisfy the linking step
  - E.g., to use the C math library, use `#include <math.h>` and `-lm`
- GCC internals (three components)
  - **Front end** converts C/C++/Ada/other languages into a intermediate representation (IR; GCC uses one called GIMPLE)
  - **Middle end** optimizes the IR, remove unnecessary code, etc.
  - **Back end** emits machine-specific code (x86, AArch64, MIPS, etc.)
- Know what GCC is useful for (aside from compilation)
  - security improvement (stack-protection features)
  - performance improvement (optimization flags)
    - also performance analysis (can profile using `gcc --coverage`)
  - static checking (e.g., `_Static_assert( )` in C; `-W` enables compiler Warnings)
  - dynamic checking (sanitizers)
- Know about *defensive programming* techniques (see lectures 11/24, 12/1)
  - traces and logs (print statements)
  - checkpoint/restart
  - assertions
  - exception handling (`try … catch …`)
  - barricades
  - interpreters and virtual machines
- Tools
  - GNU Debugger (GDB): step through code and inspect as you go
  - Valgrind memcheck: checks memory leaks, usage of uninitialized memory, etc.
    - Similar to AddressSanitizer but slower and more thorough

- ImpossibleWare, Inc. claims that they have created a new C compiler, IWCC, that can detect *all* possible bugs in any C program, including stack overflows, double frees, out-of-bound reads, and logic errors. Unsurprisingly, they are charging top dollar for this compiler software. You are an engineer at a software firm tasked to evaluate IWCC. Do you believe their claim? If so, describe some possible techniques the compiler may use. If not, provide some rationale for being skeptical.
- Consider the '`assert`' macro available in C's assert.h. In most cases, it evaluates the argument, and aborts the program immediately if the argument turns out to be 0 (indicating "false"). However, if the `NDEBUG` macro is defined during compilation, `assert` does *not* attempt to evaluate its argument or checks its value. What are some scenarios where you would want to set `NDEBUG`? What are some good reasons *not* to set `NDEBUG` *even in those scenarios*?

Version Control

Corresponding lectures: 12/3

- Know why version control is important
- Know about pros/cons for different approaches to version control
    - What data to back up
    - How often to back up
- Know strategies for more cost-effective backups, and their pros/cons
    - E.g., what level of disaster can this backup system recover from
- Some historical version control systems
    - SCCS, RCS, CVS, Subversion (SVN)
- Techniques in version control systems
    - Compression
    - Cryptographic signing (PGP)
    - Copy-on-write
    - Atomic commits
    - Distributed (Git, Mercurial, Bazaar, …) vs. centralized (Subversion, Perforce, …)
    - …
- See Spring 2020 Midterm 2 for practice questions (solutions)

Git Internals

Corresponding lectures: 12/8, 12/10

- Each Git repo contains refs and objects
- Each object has a unique **hash** (a SHA-1 checksum) that identifies it
- Refs (e.g., branches) point to objects through their hashes
    - A ref is like a movable pointer; can refer to different objects over time
    - E.g., when you make a commit on a branch, the branch ref now refers to the new commit
- Be familiar with the relevant Git objects from assignment 6
    - tree, blob, commit, …

- To display the content of a Git object, use `git cat-file -p`
- Know how to do things in Git or figure out how to do them
    - git clone, pull, push, commit, diff, show, add, rm, merge, rebase, fetch, remote, …
    - git pull = git fetch + git merge
    - git pull -r = git fetch + git rebase
- See Spring 2020 Midterm 2 for practice questions (solutions)