Software Development (CSCI2132)

Covers Introduction to UNIX to Introduction to C

Lecture Dates: January 4th, 6th, 9th, 13th, 16th, 18th, 23rd, 25th, 27th, 30th, 2012
February 1st, 3rd, 6th, 2012

Assignment Notes:

Emacs: The command M-x auto-fill-mode toggles filling on or off. It will insert a newline at a certain position, taking care to pass a word onto the next line if it would be otherwise split. At what character will it do so? Probably around 72 unless you tell it otherwise. Here's how to choose: C-u 80 C-x f sets the width (80 characters, in this example) of your paragraph but does not reformat the paragraph. M-q reformats the paragraph.

Introduction to UNIX, Software Development

Reading Material: *UNIX (Chapter I), Chapter II (up to p. 26 – 44 ... 50 ... 69-75 (emacs)), Chapter III (114-115, 151-154).*

What is ***Software Development***?

- Writing large computer programs
  - Systems consisting of large number of ***modules*** (smaller programs).
  - Often written by *different* programmers.
  - Specific techniques.
    - Software development processes.
    - Source code management.
    - Software testing, debugging.

- Understanding how systems work on the low level.
  - ***High level***: Closer to the users, high-level abstraction. Languages easier to use, closer to our language.
  - ***Low level***: Closer to machine language, how machines work. Will specify *sequence of operations*.
  - Supports above; would like someone to design a car without understanding how it works? Same applies here. *Expected to know how computer systems work at low level*.

- Why C?
  - A ***low-level programming language***. Closer to assembly/machine code than Java, other high-level languages.
    - Example: In Java, arrays start from 0 to n-1. *If index out of array, will be checked in runtime. In C, this checking is not done*. Supposed to know what are doing.
    - Designed to be **efficient**; *less overhead*.
  - Freedom it gives to programmers *motivate discussions of software engineering principles*.
  - *Very* widely used.
    - Systems written in C: Unix, Linux, …

        - C-based programming languages: C++ (OOP), PHP, Java, C#.

- Why UNIX?
    - Does *not* hide OS operations. Closer to low-level.
    - Linux is **open-source**. Can install; free. Unix-like system.
    - Widely used: **servers**.
- *Ultimately*, helps you become an ***effective software developer***.

Class Information:

- Office: 222 (CS Bldg); hours: 1:30-2:30 Wed, Thurs.
- Two textbooks: C Textbook, Unix Textbook.
    - Lots of exercises in C Textbook.

- Midterms: February 8, March 7.
- Optional bonus programming exam (3% bonus): During lab of March 28. (Second-last week). Will not be easy.

- Lectures
    - *Note-taking* is required.
    - *Long examples* (programs).
        - Will be projected.
        - Code will be available electronically (few comments, with blanks).
        - Will do fill-in-the-blank questions in class.
        - Notes about design, some comments will be given in class (with line #s).
        - After class, advised to fill in the blanks, add comments, run them on bluenose server, print them to study them.
        - Try to print them before class.

- Assignments
    - Posted on Thursdays.
    - Due at 5pm on due dates (also Thursdays). Designed to be weekly.
    - Exact dates on course information sheet.
    - Difficulty ratings vary.
        - 3 bronze
        - 3 silver
        - 2 gold
        - Small number of bonus marks are available on gold assignments.
    - Marking scheme (programming assignments).
        - Correctness.
        - Design.
        - Documentation.
    - Correctness.
        - Automatic testing program.
        - Similar to client evaluation of software product. Will see if features are implemented

correctly. Not (just) going to look at code.
        - Failing to pass more than half of test cases will affect design, documentation marks as well.
        - When your program is incorrect, does not compile, does not work:
            - **DO**:
                - Debug.
                - Try to mark your program run for at least simpler cases.
            - **DO NOT**:
                - Keep writing without testing.
                - Not written assignments.
        - Exams are written, so partial marks may be given there.

- Programming Environment: ***Labs***
    - ***Eight*** mandatory labs.
        - Course materials that are more suitable for lab work than classroom learning.
        - Helps to get ready for some assignments.
        - Last lab will give practice on material not covered by assignments.
        - Are on course information sheet.
    - No labs (on):
        - Today
        - Study week
        - March 28 (bonus programming exam)
    - (Three) other labs:
        - Assignment
        - Practice questions
    - In the lab: SSH, Server: bluenose.cs.dal.ca
    - At home: SSH, work on Linux PC directly.

- Course website.
    - http://web.cs.dal.ca/~mhe/csci2132/
    - Corrections, etc.
    - Practice questions.
    - Check announcements regularly.

- Last, but very important.
    - Historically, many people fail.
    - Why?
        - Computer science is not easy.
        - This is one of the first courses that requires students to spend a lot of time studying.
    - Suggestions
        - Start working on assignments early (remember the client evaluation of software products).
        - Study after each lecture, *practice* (multiple ways to do things in C – have to know why).
        - Ask questions if there is anything you do not understand.

- **UNIX**, Introduction to
  - Operating system.
    - ***Onion skin model***. See handout. Application programs at the top, then operating system(s), then hardware in the middle.
    - (Some) functions of an operating system:
      - The level between application programs and hardware; ***manages hardware resources*** of the computer system.
        - Central Processing Unit (CPU) time.
        - Memory access.
        - Secondary disk space.
        - Mouse, keyboard.
        - DVD, CD drives.
      - Provides an ***interface between application programs and hardware***.
        - Easier for the application to manipulate the hardware. *Hides the complexities* of the hardware interface from application programs.
        - Protects the hardware from user mistakes and programming errors (to prevent crashes).
          - Example: segmentation fault runtime errors – when run program, program stops without generating output: one common cause is in C have the freedom to do low-level program – can write to different memory addresses.
          - If write to memory address of where program is being written – OS stops program.
      - ***Protects users' programs and data from each other*** (security issue).
        - File permissions.
        - See who can access, read/edit files.
      - ***Supports inter-process communication*** (communication between processes).
        - Example: output of one process can be used as the input for another.
  - History
    - See why it was designed, how it has evolved, philosophy behind the system.
    - Has a long history.
    - Created in 1969. 1st Unix version by Ken Thompson at Bell Labs. Context:
      - Hardware was small (capacity), slow at this time.
      - Users were programmers – not necessarily wanting features that were easy to use.
      - Planned to create a game called Space Wars (operating system was limited at the time – MULTICS – did many different things). Called an operating system of his own: UNIX (does one thing well – UNI; gave performance he needed to write his game).
      - Initially written in assembly language (ASCII versions of the machine instructions).
        - Very hard to understand.
        - Very hard to code.
    - In the early 70's, decision made to rewrite Unix in C by Thompson and Dennis Ritchie.
      - Big advantage. At that time, most OSs written in assembly language because it was efficient and since hardware was slow.
      - C, at that time, was the only non-assembly language that was also efficient. Became

first OS for which code was very understandable.
        - More programmers had the ability to actually change code, contribute to it.
        - Because of this, became very popular.
        - Also makes the OS more portable; assembly language is different for different platforms but C language is same across platforms (just need different compilers).
        - Small portion still written in assembly language.
        - More people began to contribute to the project.
    - Lots of researchers started to contribute in 1970s from academic institutions.
        - AT&T Bell Labs licensed the OS to many universities free of charge.
        - Particularly graduate students from UC Berkley.
        - Created a distribution of the system called BSD.
        - University of Washington: Pine
    - 1980s: Now there were so many distributions, versions, variants. Even though they share same philosophy, ancestor, not all of them are entirely compatible. UNIX wars.
        - Who is the standard? Primarily between two: System V (5, AT&T) vs. BSD UNIX (Berkley Standard Distribution from UC Berkley).
        - No real winner. System V became more popular later on, though.
        - Modern UNIX systems like Solaris, Sun have features borrowed from both (strengths, weaknesses in both).
    - 1985: Free software created based on UNIX. Free software foundation launches GNU project (recursive acronym = GNU's Not UNIX).
        - Wanted to create a complete "UNIX-like" system *except for the kernel* (part of UNIX directly controlling the hardware). This didn't really work; licensing problems for the kernel.
        - Wanted to write their own code because of licensing problems. Wrote compilers, e-mail programs, utilities, etc.
    - 1991: Not completely open-source because of the kernel. Linus Torvalds announced the Linux project.
        - It's the kernel. Open-source (free, people can change the source just like the GNU project) UNIX-like OS kernel.
        - Does not share code with UNIX.
        - Share same design philosophy, same way managing resources, etc.
        - Technically not a branch, but so close cannot tell the difference.
        - Usable in 1992.
        - So now people just put GNU and Linux together to create GNU/Linux. Technically people call "Linux" an OS, it technically refers only to the kernel.
    - Most webservers now use Linux. More than 40 years old – so why is it still popular despite its age? Many reasons, but one is the design philosophy.
- UNIX Philosophy
    - At first, it was about efficiency; system that did one thing well.
    - Philosophy for ***Utilities***:
        - In the UNIX system, a program should do *one thing*. Example: a command `ls` lists what files you have in the current folder. `who` lists users.
        - It is therefore easier to make them do this one particular task *well*. It should do it well.

        - Complex tasks should be performed using these utilities *together*.
    - ***Pipes***
        - Used to specify the output of one process to be used as the input of another.
        - Can use pipes to chain commands together to perform complex tasks. Create something called a *pipeline*.
        - *Pipeline*: [Process 1] → Data → [Process 2] → Data → [Process 3] → Data → …
        - Example: `who` used to list names of users currently online. But also want to sort them alphabetically. `who | sort`.
- *Aside*: Using "Pseudo-Terminals" or virtual terminals in lab on Mac or Windows. Get a unique ID (integer) which how the who command sorts users.
- Some Notable Features of UNIX
    - Allows many users to access one computer system at the same time.
        - CPU time is divided into slices, allocated to different users. Memory divided into pages; disk space into blocks.
        - Lets users share these resources.
    - It is portable. Mostly written in C.
        - Available in a wide range of platforms.
        - Might not be using PCs, Macs, etc.
- Getting Started
    - Architecture – Going into more detail: seeing the architecture.
        - Diagram – in centre is still the hardware. Then are the UNIX layers (inner: kernel). See handout.
            - *Kernel*: Hub of the OS. Manages software resources in relation to the hardware – allocates time and memory to programs. File storage. Communications in response to system calls.
            - *Shell*: Text-based interface to UNIX system. Command line interpreter. Interprets user commands and arranges for execution. For example: typing in who. OS has to understand what you are doing. Shell sees it and looks for actual executable file, etc.
            - *Commands and Utilities*: Use the kernel to do a task. Example: who command.
            - *Library Routines*: For programmers. APIs, for example.
            - *Application Programs*: Use everything. The outer-most layer.
    - Logging In
        - Have an account (CSID, for example).
        - Windows: Download, install Putty. Enter servername, username, password.
        - Mac, Linux: Open a Terminal, type in ssh (secure shell protocol). For example: `ssh username@bluenose.cs.dal.ca`
    - Shell
        - `username@bluenose.cs.dal.ca:~$`
        - Will see something like this. Displayed by the shell.
        - The $ sign gives you an idea of what kind of shell is being used. Sometimes see %. We'll talk about this later.
    - Running a Utility
        - Simply enter the command (name), press enter.
        - Examples

        - `date` (ex: *Fri Jan 6 11:22:30 AST 2012*)
        - `clear` (clears the screen)
        - `passwd` (change the acct password; text-based form)
    - Input/Output (I/O) and Error Channels
        - Three channels. One for input, output, error messages.
        - By default, I/O channels are `stdin` (standard input), `stdout` (standard output), `stderr` (standard error).
        - A program reads input from stdin by default. Writes to stdout. Writes errors to stderr. These are names given by the OS.
        - By default, all three channels are the terminal running the command/program. Think of it as containing a keyboard, monitor.
        - Can redirect, change these channels.
    - Getting Help (Online)
        - The `man` command.
        - Gets a UNIX manual page about a command or system function.
        - Useful for both users, programmers.
        - Two main ways to use this command.
            - Explicitly: `man commandname`. Example: `man clear`.
            - By keyword: `man –k keyword`.
    - Special Characters
        - Metacharacters
        - There is a list of them; see textbook. Also a command to get them.
        - Two of them (most important to most users);
            - Terminating a program: Ctrl-C (^C)
            - End of Input, FILE, EOF: Ctrl-D (^D)
                - Many utilities take input from other files, keyboard.
                - Example: reading from a file, user.
                - How the utility knows you are done with input in the latter case (from the keyboard).
                - For example: UNIX command `cat`. Outputs same lines whenever you type a line. “Copycat”. The “>” key can use to redirect the output, i.e. to a file.
                    - ***> Output Redirection***
                    - `cat > hamlet.txt` would output everything from the command to this text file.
                    - ```
                      To be or not to be
                      That is the question
                      ^D
                      ```
                    - Can also output the text from the file: `cat hamlet.txt`
    - Logging Out
        - May be tempted to close it. Not what we’re supposed to do.
        - In most systems, hitting ^D will do this. Sometimes, may think you do it by mistake.
        - Typing: `logout`, `exit` (can take a parameter and use it as an exit code).
- Files and Directories – Data Storage
    - What is a file?
        - UNIX file model is different from what we think a file is.
                      ```

        - A file is a *stream of bytes* – anything that can generate a sequence of characters or be represented as one. Example: regular files, stdin, stdout, stderr, keyboard, monitor, hard drive, CD/DVD drive. Why these redirections work. *Represent all input, output* under UNIX.
    - Abstraction: One general r/w interface works for all. Good software development practice.
    - In UNIX, *everything is a file or a process*.
    - Types of Files
        - Regular file (e.g. Text, graphics, video, source code, executable program, etc.). Denoted as a –.
        - Directory file. Denoted as a d.
        - Unbuffered special files (character devices). Do not use a buffer; work character-by-character. For example, when typing on a keyboard, in the terminal. Denoted as a c.
        - Buffered special files (block devices; block special files). Have a buffer – for example, when using a hard disk (which is slow – data has to be loaded into memory before it can be processed – done block by block), or when using a USB key. Denoted as a b.
        - Symbolic links (reference to some other file). Like creating a shortcut in Windows, alias in Mac. Learn more about this later. Denoted as an l.
        - Pipe: Similar to what we learned about earlier; here they are files. Used for inter-process communication. Denoted as a p.
        - Socket: Used for inter-process communication. Denoted as an s.
    - When in a directory, how do we know what types of files those in a directory are?
        - Using the ls -l command (long-listing, see who owns the file, size, etc.).
        - Symbols as shown above will be shown.
- The UNIX Directory Structure
    - All of the files are contained in a unique main directory called the *root* directory. (/ denotes a root directory).
    - Below that, typically there are several files and subdirectories.
        - Below root there are usually:
            - A users directory.
            - A bin directory.
            - A tmp directory.
            - A usr directory.
            - And more.
        - A parent directory contains *subdirectories*.
    - These subdirectories contain files and other directories.
        - For us, the users directory has directories: cs, faculty.
        - In the usr directory, there is usually: bin, lib, local, ...
    - We can keep doing this. *Recurse*. Faculty has directory mhe, which may have more directories, and so on.
    - This is a *tree structure*. Why the main directory is called root; root in tree hierarchy.
    - For each directory, there are two special entries.
        - Itself (.)
        - Parent (..)

- What's the parent of the root? We say the parent of the root is *itself*. The root is its own parent.

- How do we refer to files?
  - Two different files could have the same filename in different directories. Cannot have files with same names in same directory.
  - Will need to learn more about the notion of pathnames (path).
    - Even though they have same name, path from root to them is d ifferent.
    - Node-to-node movement is called a path. A pathname is a sequence of directory names that leads you through the file system hierarchyfrom a starting directory to a target file.
  - **Absolute path**: Path relative to root. Example: /bin, /usr/bin. Subsequent slashes are called "separators" as they do not denote the root directory. To a file name: /users/faculty/mhe/csci2132/lab1/HelloWorld.class.
  - **Relative path**: Instead of starting from root, start from current directory. Relative to current working directory.
    - For example, log into UNIX and start from home directory (/users/faculty/mhe). The command pwd prints "the current working directory".
    - Relative to there, you can say: csci2132/lab1/HelloWorld.class.
    - To change working directory, use command cd (change directory). For example: cd ../.. (goes up two directories). By itself, takes you to home directory.
  - Related UNIX Commands
    - Will learn some of them this afternoon.
    - ls Listing the contents of a directory.
    - cat / more / head / tail Showing the content of a file.
    - mv Renaming or moving a file.
    - mkdir Making a directory.
    - cd Moving to a directory.
    - cp Copying a file.
    - emacs / vi Editing a file. List of UNIX editors.
    - rm Deleting a file. Deleting a directory that is not empty: -r.
    - rmdir Deleting a directory (empty).
    - wc Counting the number of words in a file (text). Can also count number of lines.
  - File Permissions (protecting data).
    - Users and Groups.
      - We have an account that has an associated username. Every user has:
        - a unique username (text string)
        - a unique user ID (an integer – how UNIX internally identifies each user since operations on strings are slow).
      - Users are also grouped into groups. Every user is a member of a group. Groups have:
        - a unique group name (text string)
        - a unique group ID (an integer)
      - Users can be in more than one group.
    - Users in the same group of the user who owns a file (can) have the same

permissions for that file.

- To find out what group you are in, there is a command for this to list your groups: groups.
- File Ownership
  - When you create a file, it belongs to you. In UNIX, each file is associated with a user (its unique file owner, a username).
  - Each file is also associated with a group (file group).
- Each file has three sets of permissions.
  - File owner.
  - Users in the file group.
  - Other: everyone else.
- Each set has three different permissions.
  - Read
  - Write (Edit, Delete, etc.)
  - Run, Execute
- What does this mean for a directory? A directory is also a file. See textbook.
- Each permission can be represented in 3 bits (Yes/No for each).
  - Example: 101 applies to one of the sets. This means the set allows for reading and executing the file, but not writing (to it).
  - This means you can use 9 bits to represent all of the permissions for one file. Example: Owner-Group-Other (Read-Write-Execute) 111101000
  - So it makes to use one character for each set. An octal number of three digits can be used (since each set is from 000 to 111).
  - So, 111101000 is 750.
- Checking Permissions: Using ls with the flag -l (long-listing for files in result; contains a lot of things). Can pass in a path to a folder or file as well.
  - For example: ls -l HelloWorld.java
  - Gives you: -rw-r—r-- 1 mhe csfac 116 Jan 9 15:27 HelloWorld.java
    - The first character says it is a file (-).
    - The nine after that are equivalent to the permissions (110100100).
    - Execute would be an “x”.
    - The octal number for this would be 644.
    - The 1 is the number of “hard links” (talk about later).
    - Then is the owner (mhe).
    - Then is the group (csfac).
    - Then is the size of the file (in bytes).
    - Then is the timestamp for last modification.
    - Finally is the filename.
- Changing Permissions: chmod (Change Mode)
  - Example: chmod 664 HelloWorld.java will give the group write permissions. We call the 664 the mode.
  - There is another way to write the mode (if we want to change it incrementally; don't care about all permissions).
  - chmod g+w (grants group write permissions).
- More Commands

        - Change a file's group: chgrp
        - Change a file's owner (on some systems, only admin can do this): chown
        - Change (effective) groups: newgrp
    - Only one group is your *effective* group at any time. Anything you create will be associated with this group.
        - When you create a file from the shell, the file group is the effective group of your shell (cannot change default effective group, but can change current one).
        - Can use this to create files for different groups.
- Redirection, Filename Substition, and Links
    - ***Redirection***: The idea of changing where input, output, and errors are sent by a program (default: stdin, stdout, stderr).
    - Output Redirection
        - Stores output of a process to a file.
        - In the shell: $ command > filename
        - For example: $ ls lab1 > listing. If this file exists already, is overwritten; otherwise, it is created if it does not.
        - Can append results, rather than overwrite, by using >> rather than >.
    - Input Redirection
        - Redirects stdin to a file. Lets program read input from a file.
        - In the shell: $ command < filename
        - For example: $ mail username < HelloWorld.java. This command sends e-mail; say you used emacs to write an e-mail already. If on same server, just use the username. Otherwise, full e-mail address.
    - Error Redirection
        - This stores error messages to a file.
        - In the shell: $ command 2> filename
        - For example: $ rm x 2> error. If this file exists already, is overwritten; otherwise, it is created if it does not.
        - Like before, can append using 2>> rather than 2>.
    - File Descriptors (to stdin, stdout, stderr).
        - 0, 1, 2 respectively.
        - Therefore, 2 is the file descriptor for stderr.
        - So, another way of redirecting input is using 0<, or using 1> to redirect output.
    - Filename Substitution (Wildcards)
        - *Wildcards*: These special characters that are expanded by the shell to specify a set(s) of file. Feature provided by the shell.
        - In other words, any word on the command line that contains wildcard characters is treated as a *pattern* which allows replacing by an alphabetically sorted list of all matching filenames.
        - Types of Wildcards
            - ? - Matches any single character.
            - * - Matches any string, including the empty string.
            - [ ] - Matches any single character from the set specified, contained by the brackets. More restrictive than ?.

        - Example: [unix] – this character can be either a u, n, i, or x.
        - Example: [0-9] – this character can be any digit between 0 and 9, including both 0 and 9.
        - Example: [a-zA-Z] – this character can be any lowercase or upper character (any English alphabet letter).
        - The ! symbol can be used to *complement* a set.
    - Using these with Unix commands.
        - Example: We have a folder csci2132 with a sub-folder lab1. Want to see what Java files there are in this sub-folder. $ ls ~/csci2132/lab1/*.java
        - Another example: $ ls ~/csci2132/lab1/Hell?World.java
        - Those files that start with a digit: $ ls [0-9]*
        - Those files that do not start with a digit: $ ls [!0-9]*
        - Can work with other commands as well. For example, copy all Java files. $ ~/csci2132/lab1.bk/*.java ~/csci2132/lab1
- Unix File Representation
    - A file is represented by an ***inode*** (index). This is a data structure that contains information about:
        - file type
        - permissions
        - owners, group IDs (integers)
        - last modification and access times
        - file size; size of object being stored
        - where the data is stored on the disk
    - For each inode structure, there is a unique inode number (integer) and each file has a unique inode number.
    - A directory is therefore a file that contains a table of <inode number, filename> pairs.
    - To make all of this work, there is one system-wide inode table, for all of the files. Inode numbers are indices into it.
- (Hard) Links
    - When we create a hard link, we add an entry (a pair) into the directory file for where we want to create the link.
    - This pair contains a *new* filename, but duplicates an existing inode number so the original file is refered to.
    - We do not create a new inode. Essentially create a new filename for the file. Can also use the new filename to access the original file.
    - Command in Unix: $ ln target linkname
    - For example:
        - pwd ~/csci2132
        - ln lab1/HelloWorld.java HelloWorld.java
    - Deleting a link will not delete the original file; as long as there is a name for this file using its inode number, the file is still there. So,
    - *A file is deleted when no hard links to its inode exist*. This includes the original target file (not special, just another hard link).
    - How do you know how many links are pointing to the same file?

        - Checking hard link count: $ ls -l
        - The hard link count is present as an integer (see above).
    - Restrictions
        - In a file system, can use multiple hard drives. But we cannot create a hard link to a file stored in a different hard drive. *Can not span different drives*. To do with internal representation.
        - Would be *problematic to create a link to a directory* (would create loop in tree structure). So, cannot, typically. In some variants of Unix systems, allow root user to do this.
    - So what if we want to have a short path to a directory? Span different drives? We use soft links.
- (Soft) Links – aka Symbolic Links
    - A file that has the (absolute or relative) pathname of another file stored within it as its data (part).
    - Talking about a file that has a data part storing a pathname, so *creating a new inode* and new data for this file.
    - This new inode points to some file data and this data is a pathname which can be used to return to the target and access it.
    - Essentially this is a "shortcut".
    - Command in Unix: $ ln -s target linkname
    - Does not increase the hard link count.
    - Removing a symbolic link does not affect the original file whatsoever.
    - However, if the target file is removed (which is being pointed to by a symbolic link), the soft link still exists (since we do not keep track of it as a count). When we try to access it, we will get an error.
    - Note: ls -l displays the pathname stored in a soft link.
    - Hard Links vs. Soft Links (Behavior)
        - Deletion
        - Restrictions (can span drives, can create soft links to directories).
    - Why not use soft links for all links, then? Some programs will treat them merely as files containing pathnames (will not point to the original target). Merely treats them as files themselves. Will not be "fooled" by the shortcut. For example, the copy command, the soft link itself is copied and not the original file.

- Pipes, Regular Expressions, and Filters
    - ***Pipes***
        - The shell is the program that interprets all our commands. It allows you to use the stdout of one process as the stdin of another.
        - This is done by connecting the processes together via the pipe(|) metacharacter.
        - Therefore, a ***pipeline*** is a sequence of processes chained in this manner.
        - General steps of using pipes in solving problems.
            - Break the problem into subproblems.
            - Use a program to solve each subproblem.
        - For example, to output the number of files in the current directory: ls l wc -l
        - The above counts the number of lines in the ls output.
    - ***Regular Expressions***

- The problem: text search. Want it to be fast, flexible.
- Regular expressions are fast (DFA) – implemented naturally on computers, flexible (allows you to search for patterns).
- Regular exprssions can be defined as a sequence of characters (regular characters and/or special characters) that together specify a pattern to match against strings.
- Used in many programs.
  - grep
  - sed
  - perl
- Metacharacters are used to specify matching rules. Similar to wildcards (which were only used for file names) but here these can be used on any text strings. But recall, **these are not the same** as wildcards.
- There are two types: Basic, Extended.

- Basic Regular Expressions
  - . can be used to match *any single character*.
    - For example: a.b can be used to match aab, abb, acb, ...
    - This is already different from wildcards (? for a character).
  - [ ] are used to match any of the (single) characters enclosed in the brackets.
    - For example: c[ab]d can be used to match cad, cbd.
    - Only matches one character.
    - - is used to specify ranges, like before.
    - ^ is however what is used for the not character, rather than !. Example: [^]
  - * matches 0 or more occurences of the character that precedes it.
    - For example: a* matches the empty string (“”), “a”, “aa”, “aaa”, “aaa...”
  - ^ (carrot) when outside the [ ] matches the beginning of a line.
  - $ matches the end of a line.
  - \ inhibits the meaning of a special character. (\$ talking about the $ character).
  - For any of these(. , * , ^ , $, and \), when put inside the [ ], they lose their special meaning.
    - Only exception is the ^ character; if first character, it will act as a not in the square brackets.
    - Somewhere in the middle, is a regular character.
  - Some of these characters have special meanings in the shell, as well. So, in order to have them behave as they are here, we use ***quoting*** (' ').
- BRE Examples
  - To find one or more (space) characters we use '__*' (two spaces indicates one or more rather than 0 or more; the _ characters indicate a space and not an actual _).
  - To find empty lines, we use ^$
  - Formatted dollar amount ($23.45): \$[0-9][0-9]*\.[0-9][0-9]
- Filters
  - A type of program that:
    - Gets most of its data from stdin.
    - Writes its main results to stdout.
  - Often used in pipelines (as elements).
- grep

- A filter.
- Works with basic regular expressions. Does text search.
- Scans a file and outputs all **lines that *contains* a given pattern** specified by a regular expression.
- Syntax: grep [options] BRE [file]
    - If do not provide a file, will wait for you to input (i.e. Can be used in pipeline), line-by-line.
    - Example: Have the following text.

      Price:

      Chocolate $1.23 each
      Candy $.56 each
      $278.00

      Shirt $44.00
      $44.00

    - We use: grep '\$[0-9][0-9]*\.[0-9][0-9]' price
    - We get lines 1, 3, 5
    - Another example: File on bluenose that contains all English words. We want to get all 5-letter words that start with a or b and end with b.
    - We use: grep '^[ab]...b$' /usr/share/dict/linux.words

- Reading Material
    - Pipes (154-155), BRE (665-666), grep (84-87)
    - In lab, are using: uniq/sort (87-91)
    - For next time: ERE (667-668), UNIX shells (145-162)
- Regular Expressions, continued
    - ls *.java is equivalent to ls | grep '\.java$'
    - Looking for all files in /bin that contain exactly one '-': ls /bin | grep '^[^-]*-[^-]*$'
    - Wildcards can work with other commands where you cannot use grep because grep simply does text search and not anything else.
    - Furthermore, wildcards are part of the shell (treated differently) so do not want to use too many special characters where we'll have too few to specify filenames.
    - grep Options
        - -n asks the utility to output lines found preceded by line numbers.
        - -i asks the utility to ignore case.
        - -v asks the utility to output lines that do **not** match.
        - -w asks the utility to find only lines that contain at least one *word* that matches the exact pattern. Restricts matching to whole words only.
    - Extended Regular Expressions
        - More metacharacters.
            - + matches one or more occurences of the single preceding character.
            - ? matches zero or one occurences of the single preceding character.

        - | acts as an OR operator. For example: ERE1 | ERE2 (Extended Regular Expression 1 OR 2 have to match).
        - ( ) if you place an ERE in brackets, you may use *, +, ? to operate on that entire expression; i.e. Considers it a "single character".
    - Examples
        - Formatted dollar amount: \$[0-9]+\.[0-9][0-9]
        - One or more occurences of the string abc: (abc)+
    - Syntax for REs vary between systems.
        - This means there are other special characters on other systems that may not work on others.
        - What we learned should work on the *majority* of systems.
        - The man re-syntax command displays the user manual for regular expressions.
- fgrep/grep/egrep
    - fgrep is intended for *fixed* strings. Does not contain specific characters; less flexible – is very fast.
    - grep is the same as before. By default, works with BREs. Can specify it to work EREs (an option).
    - egrep is intended for working with EREs to do text searching.
- Shells
    - Why is it seperate from the **kernel**?
        - A program in UNIX. Therefore, if something goes wrong, our system is still up (will not go down). For instance, if running on a webserver, the webserver will stay up.
        - If want to add new features to the shell, all have to do is write a new program rather than rewrite the OS altogether; they can be replaced without rewriting the kernel. This is good practice.
    - Functionality
        - Has quite a number of them:
            - Built-in commands,
            - Scripts (will learn this later after C),
            - Variables (***local*** and ***environment variables***),
            - File redirection,
            - Wildcards,
            - Pipes,
            - Sequences
                - Can write multiple commands on one line and have them execute one after another.
                - ***Conditional*** and ***Unconditional Sequences***
            - Subshells
                - Notion of shells creating subshells.
            - Background processing.
                - If use one terminal, can do something in the foreground while having a process running in the background so do not wait for it to finish.
            - Command substitution.
                - Replace a command by its output.
                - Will explain later.

- Popular Shell Programs
  - Four popular shells:
    - **Bourne** (/bin/sh) – oldest amongst four shells, but still not first Unix shells (Thompson made a Thompson Shell).
    - **Korn** (/bin/ksh)
    - **C** (/bin/csh)
    - **Bash** (/bin/bash) – **B**ourne **A**gain **Sh**ell
      - We will be using this.
      - Backward compatible with Bourne shell.
      - Bluenose works on Bash; the /bin/sh is a softlink pointing to the Bash program.
      - Most useful features.
      - GNU licensed. Open source.
  - chsh changes the login shell; asked for password and choose pathname for what shell you want. This, however, does not work on all Unix systems.
    - Sometimes have to ask the Admin to change your login shell.
    - If you do not want to ask him, just change it everytime you login (run the shell).
    - Can also edit a config.
- Commands
  - Two types.
  - Built-in commands.
    - Commands that a shell recognizes and executes internally (code is part of the shell).
    - Therefore, very fast to execute (no overhead to load them in memory).
    - But, if we want to change them, have to change your shell. Difficult to replace.
    - Makes sense to keep a limited number.
    - cd changes current working directory.
    - logout logs out of server.
    - echo displays all arguments to stdout.
      - Example: echo 'Hello World'
      - Can also be used to display values of variables.
      - Output function for scripts (like a println).
  - External commands; not built-in.
    - Executable programs stored in the directory hierarchy.
    - Seperate from the shell, installed somewhere.
    - Overhead for loading them, but easy to replace.
    - ls,sort,grep,etc.

Readings: ERE (667-668), Shells (145-150), Variables (163-165), Processes (167-172, 230-234), Quoting (166)

Assignment #2 is online. Can work on all questions but Q1(b).
Assignment #1 solutions posted online (user: 2132, pw: TestDebug)
Midterm #1 (Feb. 8) is in-class, in a different classroom – more details this Friday.

- Variables
  - Behaviors of shells are influenced by several *string-valued* shell variables.
  - Assigning a value to a variable: VARIABLE=value (no spaces)
    - If exists, assign value.
    - If does not, creates and assign.
  - How do we know we are calling a value rather than a file? Use a metacharacter; the $ sign. Expands the value of the variable.
  - Recall the echo command. Allows us to see the value of a variable, display it.
    - Example:
      - $ firstname=Meng
      - $ lastname=He
      - $ echo $firstname $lastname
  - This will be useful when we use Shell scripts.
  - There are built-in variables (already in shell); this will affect behavior of the shell. E.g.
    - $ echo $SHELL – outputs the pathname to the shell program. Can find out what shell are using.
    - $HOME – stores the pathname to the home directory (absolute path).
      - If we do not supply an argument, the cd command changes the directory to your home directory;
      - This is essentially the same as cd $HOME or cd ~
    - $PATH – stores a list of directories (absolute or relative paths) that the shell searches for to identify external commands (e.g. Binaries – files that are executable).
  - When we run a command without a full pathname (to it), the shell will first check whether this command is a built-in one or not, and if not, then it searches the direcotires whose paths/names are stored in $PATH.
- Command Substitution
  - Using the result of a command; substitution a command by the result of its execution. Using the ` key (the left accent symbol). So: `command`
  - Shell executes the command and inserts its output in the command line.
  - Example: $ echo There are `ls | wc -l` files in the current directory.
- Quoting
  - Used to inhibit the shell's wildcard replacements, command substition, and variable substitution.
  - Single quotes (') inhibit all three.
  - Double quotes (“) inhibit wildcards only.
  - When quotes are nested, only the outer quotes have any effect.
  - For example,

```
$ ls
HelloWorld.class HelloWorld.java
$ echo 3 * 4 = 12
3 HelloWorld.class HelloWorld.java 4 = 12
```

```
$ echo “3 * 4 = 12”
3 * 4 = 12
$ echo '3 * 4 = 12'
3 * 4 = 12
$ echo “There are `ls ~ | wc -l` files in $HOME”
There are 5 files in usr/faculty/mhe
$ echo 'There are `ls ~ | wc -l` files in $HOME'
There are `ls ~ | wc -l` files in $HOME
```

- Processes
  - Programs and Processes
    - A program is a piece of code – inactive.
    - A process is a running program – active.
    - A process is instanciated from a program. Like the Class-Object Relationship from OOP.
  - Process Composition
    - (Requires) memory space, composed of 4 parts.
      - A part for the Code (contains machine instructions from the program).
      - Data (not all of the data used in the program; merely static data – values used and maintained throughout execution).
      - Heap (dynamic allocation; variables do not know until we run the program – memory has to be allocated dynamically).
      - Stack (temporary data; local variables in a function, etc.; first-in, last-out – a frame which servers the data and when we're done with the function call, we remove this frame and this part of memory can be used for function calls in the future; know sizes here vs. Heap).
    - Thread of Execution
      - Active part of the process.
      - Start from first instruction, execute it, and then so on. Sequential order.
      - Forms a thread of execution (sequential execution of program instructions).
    - Process Control Block
      - Maintained by OS.
      - In the OS, information stored about each process in a PCB. Each process has one. This includes:
        - Present position in the thread of execution (which instruction is being executed).
        - Resources allocated to the process (memory, open files, etc.).
        - Process permissions (related to user running the program to create the process).
  - Process Identification
    - Each process is identified by a unique (non-negative) integer (***PID***; process identifier).
  - Process Creation
    - Can only be created by another process.
    - When a user runs a program, the shell is what creates the process (which is a program, a process in itself) of execution.

Reading: Variables (163-165), Command Substitution (156-157), Quoting (166)
Next Time: Processes (167-172, 230-234), C Textbook Chapter 1

- A parent process "fork"s itself into two processes, itself and a child process.
- The child process is a clone of its parent (memory, content all the same).
- The child process will "exec"ute" a new program that replaces the child's original program.
- The new program begins executing inside the child.
- "Spawning" is what is used in Windows when a new process is created from scratch. Copying memory, however, is much more efficient. That is not to say spawning does not have its own advantages (it is cleaner, easier to handle).

- Foreground Processes and Background Processes
  - ***Foreground process***: a process that controls the keyboard of a terminal. Takes our input.
  - ***Background process***: Cannot read from the keyboard, from terminal. Can, however, output to the screen. When we create a process, created by suffixing an ampersand (&) to the command. Will still show output; but you can redirect stdout, stderr which may be preferable.
  - Command find is given a directory which it will descend into and you can find a file by name.
    - find / -name gcc >result 2>error &
    - Looks for gcc in root, but may have errors output because of permissions of directories. If run in background, cannot really do work in the foreground without redirection. So redirect.
- Process Control
  - The command ps displays a list of processes that are created by the current shell. Will not show system processes, other shell processes. Will show (PID,CMD,...) process ID, command used, how much CPU time has been used so far, etc.
  - To stop a process, use the command kill PID. Terminates the process with process ID PID.
- Job Control
  - A facility that allows users to do a lot of things with processes:
    - Start processes in the background (&).
    - Send processes between foreground, background. For instance, if forgot to use the &.
    - Suspend, resume processes.
  - Jobs are not just processes (are some of them). Jobs are processes under the influence of a job control facility (those in the background, those suspended).
  - Displaying a list of jobs: jobs
    - Lists all jobs and their status.
    - Status: running (in the background), stopped (suspended), ...
    - Running jobs gives us the job ID (not a process ID; an integer starting from 1,2,3,..), status, command (used to create the job).
    - Means a job also has a process ID. Use jobs -l which will display the job ID

after the process ID. Rest is the same.

- Suspending and Resuming
  - Suspending a process: ^Z on a foreground job.
  - Resuming a process.
    - As a foreground job: fg %jobID
    - As a background job: bg %jobID
  - Bringing a background job to the foreground also uses fg %jobID.
  - Bringing a foreground job to the background requires two steps:
    - Suspend it.
    - Resume it in the background.

- Introduction to C
  - Background
    - Originally invented to write OSs and other system software.
    - Before C was invented, OSs were written by assembly language (no higher-level languages that were efficient at the time).
    - C optimizes for machine efficiency (time/space) at the expense of increased implementation and debugging time.
    - When Java was first developed in 1995, were 20 times slower than C programs – now a lot more efficient, but still easy to write C programs a few times faster than equivalent Java ones.
    - A central difficulty in C is *memory management*. Programmers must do their own. (Languages like Java does this on its own; garbage collection either slows down program or does not really de-allocate memory efficiently or effectively).
    - Unlike Java, assumes programmer knows what they're doing. For example, when we create an array and access an element of the array, no checks to see if out of range, bound. If we do not do check, program is faster but there may be unexpected behavior.
  - Writing a simple program.
    - The typical Hello World program: hello.c

```
#include <stdio.h>

int main(void) {
        printf("hello,world!\n");
        return 0;
}
```

    - The first line include sinformation about C's standard I/O library.
    - The "main" function is specified as having no parameters (void).
    - The printf function is one from the standard I/O library to produce formatted output.
    - The 0 being returned is the exit code (no error, successfully).

Readings for next lecture: C Chapter II.
Midterm I Preparation: Some information posted online.
Time: February 8th – 47 minutes (10:37 to 11:24). Will be in Rm 1014 (Kenneth C. Rowe)

Cheat Sheet allowed – letter size, both sides; ID.
Material to study: Notes in class (very important) including a small amount of material explicitly left for you to read textbook.
Labs 1-4 (emacs excluded), Assignments 1,2,3, Last year's midterm, Written exercises in the C textbook those on practice page first), Textbooks (section given in the exam page).

- From Text to Executable
  - Three Steps
    - ***Preprocessing*** (by a preprocessor): modifies the program by obeying directives (commands in the source code that begin with a #).
    - ***Compiling*** (by a compiler): translates the modified program into object code (machine instructions).
    - ***Linking*** (by a linker): combines the object code with additional code needed to yield a complete executable program (e.g. Linking code together from multiple files).
  - Shell Script: Similar to scripting languages like Python or Perl. Do not have to be compiled (interpreted); easier to modify but is slower.
- The general form of a simple program.

```
directives

int main(void) {
        statements

}
```

- ***Directives*** are commands intended for the preprocessor, begin with a # sign, are one line long, and does not terminate with a semi-colon.
  - Example: #include <stdio.h>
  - The above specifies that the information in the stdio.h file should be included in the program before compilation.
  - stdio.h is an actual file (considered a *header* file - .h)
    - Can use find to find this file by name. We would get: /usr/include/stdio.h
    - We can see that C and Unix are closely integrated; most Unix distributions have this file here.
- The int main(void) is a ***function***.
  - Which are building blocks of a C program; from which they are constructed.
  - A function is essentially a series of statements grouped together and given a name.
  - printf is a ***library function*** – functions provided as part of the C implementation. Can use them in any C program as long as the appropriate header file is included.
  - The ***main function*** is the function that is called automatically when the program is executed.
    - As written in the code above, it returns an *integer* value (***exit code*** **–** 0 success, 1 error by convention),
    - and takes in *no arguments* from the command line.

- ***Statements*** are commands to be executed when the program runs.
  - Must end with a semicolon.
  - Directive is not considered a statement.
- Printing Strings: printf can display a string literal (sequence of characters enclosed in double quotes).
  - Newline character "\n" is an escape sequence (like in Java). Prints a newline.
  - Which means it does not automatically print a newline character; automatically advance to next line. Different from Java's println; similar to the print function.
  - Example: printf("hello,"); printf("world\n"); will be equivalent to the above.
  - Example: printf("hello,\nworld\n"); prints hello world on two different lines.
- Commenting:
  - /* comments */ (one or more lines)
  - Often put additional stars on the side to make this more readable, even if you don't need them.
  - Standards were created for C (since were different standards and compilers) starting with C89, and then C99 is the newest standard. One feature added in C99 is a new style comment (from C++): // comment
- Variables
  - Types
    - Each C variable must have a type assigned to it. Specifies what kind of data can be held.

      Readings for next lecture: C Chapter II (2.1-2.3, the rest), III.
      Assignment 3 is online; can work on Q1, Q2.

  - Example: int (Integer), float (Single-Precision Floating Point)
  - Declarations:
    - Like in Java, must be declared before use.
    - Syntax: type name; (e.g. int height;, float profit;)
    - In C89 or earlier, in any function, all the declarations must precede statements. In other words, two parts of a function: declaration and statements. Also applies to main function.
    - In C99, no such restriction as long as you declare it before you use it (less error-prone, makes code more readable).
  - Initialization
    - Most variables do not have default values (when declared).
    - Like in Java, can declare and initialize in one step.
    - For example: float profit = 1030.56f; (Is optional in C).
- Some Basic Operators: = (Assignment), +, -, *, /, % (mod, remainder).
- Printing the value of a variable.
  - We saw the printf function. We use it here as well.
  - Printing an integer: printf("Height: %d\n", height). Use a placeholder (e.g. %d for an integer).
  - Printing a floating point: printf("Profit: %f\n", profit). By default, will print 6 digits after the decimial point. To change this: %.2f will print 2 digits after the decimal.

- Do not need to put a variable as the second argument; can put an expression or value.

- Reading input uses another function: scanf
  - Reading an integer: scanf("%d", &height); The & symbol is important here – hard to explain what it means for now. Has to do with pointers.
  - Reading a float is similar (use %f).
- (Defining Names for) Constants
  - Done using macro definitions. One type of directives (like include).
  - For example: using the value of Pi. #define PI 3.14159f
  - Recall: preprocessor handles directives, modifies source program (still in text format). When modifies it, will make use of this directive by replacing all occurences of the name (each macro) by its value.
  - NOTE: A macro definition does **NOT** declare a variable. Have to be careful. Compiler will not check its type.
  - The value of the macro can also be an expression (does not have to be a single value). For example: #define RECIPROCAL_OF_PI (1.0f/3.14159f)
  - Can also use other macros in the definition of a macro as long as defined before.
  - Convention in C: using all capital letters. Makes it easier to distinguish in the code.
  - If the expression contains operators, place within parantheses.
    - Reason: Say we wanted to declare a variable float pi = 1 / RECIPROCAL_OF_PI
    - When the preprocessor passes through the file, this becomes: 1 / (1.0f/3.14159f)
- Identifiers
  - Names for variables, functions, macros, etc.
  - May contain: letters, digits, underscores.
  - Must begin with: a letter, or an underscore.
- Precedence rules are about the same as Java.
- Program Example: Given a price before HST, and the payment in dollar amounts, calculate the balance to be returned.

```
#include <stdio.h>
#define HST 0.15f

int main(void) {

        // Declarations.
        float price, payment, balance;

        // Input.
        printf("Enter the price: ");
        scanf("%f",&price);
        printf("Enter the payment: ");
        scanf("%f",&payment);

        // Calculations.
        balance = payment – price*(1+HST);

        // Output.
```

```
        printf("Balance to be returned to the customer is %.2f", balance);

        return 0;

    }
```

- More about Formatted Input/Output (I/O)
  - Output: printf
    - The f means formatted.
    - Syntax: printf(string,exp1,exp2,...)
    - String and list of expressions. The string is known as the "format string". For each placeholder, there is an expression.

      Practice Question: Project 6, Chapter 2. (Will be online).
      Written Exercises: 2,3,5 in Chapter 2. (Will be online).
      Assignment Q3. Reading for next lecture: Chapter 3 (C).

    - A format string is a C string that may contain ***conversion specifications***.
    - Conversion specifications are the placeholds to be filled in during printing (those shown above).
    - The general form of a conversion specification is: %m.px (m,p are integer constants which are optional; when p is not present, we omit the dot – x is always a single letter, such as d or f).
    - To print %: use %%
    - Escape Sequences: \n, \t (tab), \a (bell), \\ (backslash).
  - Input: scanf
    - Reads input according to a specified format.
    - Format string often contains conversion specifications *only*. For example:

```
int i,j;
float x,y;
scanf("%d%d%f%f"), &i, &j, &x, &y);
```

    - Above expects the user to input four values. For example: user inputs 1 -20 .3 -4.0e3
    - Remember, the & here is very important. Usually required (but not always).
    - How scanf works is by pattern matching.
      - For each conversion specification in the format string, scanf tries to locate an item of the appropriate type in the input data.
      - Skips white space characters (spaces, newlines, tabs) if necessary.
      - It then reads the item, stopping when it encounters a character that cannot possibly belong to it.
      - If the item was read successfully, continue.
      - Otherwise, return immediately.

- For example: 1-20.3-4.0e3
  - Minus cannot be part of an integer, stops after 1 (which is assigned to i).
  - The value assigned to j is -20. Once it sees the dot, it will stop.
  - The value assigned to x is .3.
  - The value assigned to y is -4.0e3.
  - So it happens this is *still* correct.
- The final new line will be the 1st character read by the next scanf (put back).
- Another example: 1 -20.3 -4.5 5.5
  - Value of i: 1
  - Value of j: -20
  - Value of x: 0.3
  - Value of y: -4.5
  - The 5.5 will not be used for anything; it will be put back for the next scanf.
- Or: 1.1 -20 -4.5 .5
  - Value of i: 1
  - Second value cannot be read successfully. scanf will return (variables will not be initialized). This is dangerous.
- Ordinary characters in format strings.
  - White-space characters.
    - When scanf encounters one or more consecuritve white-space characters in a format string.
    - Repeatedly reads (zero or more) white-space characters from stdin.
    - Keeps going until reaches a non-white-space character. This non-white-space character is "put back" so it can be assigned to the next value.
    - Another case of pattern matching.
  - Other characters.
    - A non-white-space character in a format string is compared with the next input characters.
    - If they match, both will be discarded and scanf continues.
    - Otherwise, this input character is put back.
  - Example:

```
int i; float x;
scanf("%d &f",&i,&x);
scanf("%d%f",&i,&x);
```

    - These two function calls are equivalent.
    - For example, if user inputs 10.5 (10 will be assigned to i, 0.5 will be assigned to x).
  - Example:

```
scanf("%d ",&i);
scanf("%d",&i);
```

    - These two function calls are *not* equivalent.

        - The first will *wait* until you input a non-white-space character after you input the value and then return.
        - The second will not.
    - Example:

```
float x,y;
scanf("%f,%f",&x,&y);
scanf("%f ,%f",&x,&y);
```

        - These two function calls are *not* equivalent.
        - For an input such as 5.5 ,6e5 (x = 5.5 for the first scanf, x = 5.5 and y = 6e5 for the second).
- Example: Adding Fractions
    - Input: 2/3+1/6 (need to put whitespace characters in format string for scanf if want to allow spaces)
    - Output: 15/18
    - Using formula: a/b + c/d = (ad+bc)/bd

```
#include <stdio.h>
int main(void) {
        int num1,denom1,num2,denom2;
        int result_num,result_denom;
        printf("Enter expression: ");
        scanf("%d/%d+%d/%d",&num1,&denom1,&num2,&denom2);
        result_num = num1*denom2 + num2*denom1;
        result_denom = denom1*denom2;
        printf("The sum is %d/%d",result_num,result_denom);
        return 0;
}
```

- Operators, Expressions, Statements
    - Very similar to Java in a lot of ways. Almost identical in some ways.
    - Arithmetic Operators
        - Addition, Subtraction, Multiplication,
        - Division: Different between int/float (like in Java).
        - Remainder (Modulo): Operands must be of **integer** type (int or similar we will learn in future). Different than Java.
    - Operator / and Implementation-Defined Behavior
        - When either operand is negative (implies result will be negative), do we round up or down the result?
        - For example, (-10)/7 = -1 or -2?
        - ***Implementation-defined behavior*** is the idea that C standard does not define everything (leaves parts of its language unspecified deliberately).
            - Compiler on a certain platform fills in these unspecified parts.
            - An "implementation" is a compiler on a specific platform.

        - In most cases, this *is for efficiency*.
    - Division
        - C89 or earlier: Implementation-defined. In some CPUs, rounded up; others, rounded down (doing otherwise requires more instructions which may including jumping – not just using one machine instruction). Therefore, compiler chooses based on platform.
        - C99: Result is always rounded towards zero. No longer implementation-defined.
            - For example: (-10)/7 = -1
            - But why would do this? By this time, most CPUs all do this. Trade-off between efficiency and probability. Do not want to sacrifice consistency.
            - Therefore, do not have to worry about getting a different result on a different platform.
- Assignment Operators
    - =, +=, -=, /=, %=
- Increment/Decrement Operator
    - ++,--
    - Different when used as prefix or postfix. ++i vs. i++ will have different effects.
    - So, i++ will first return the value before increment and then increment, while ++i returns it after.
- Expression Evaluation
    - Precedence
    - Associativity (right-to-left for example with ------i).
    - ***Order of Subexpression Evaluation***
        - The rules of precedence and associativity allow us to break any C expression into subexpressions.
        - They do not always determine the value of expressions, which may depend on the order in which its subexpressions are evaluated.
        - For example: a = 5; c = (b=a+2)-(a=1); If evaluated left-to-right, c = 6. Otherwise, c = 2.
        - Another example: i = 2; j = i*i++; Left-to-right, j = 4. Otherwise, j = 6.
        - In C, the order of subexpression evaluation is an example of **unspecified behavior**.
            - Unspecified vs. implementation-defined: An unspecified behavior is not required to be documented by the implementation. Usually, compiler is free to choose whatever is most efficient.
            - Error in textbook: says this is undefined behavior. This is not right (in C, often used to talk about error-handling). C does not define how to handle some of the errors. For example, division by 0.
        - So how do we avoid this? Rule: when the order matters, just use multiple expressions rather than just 1. For example: a = 5; b = a + 2; a = 1; c = b – a;
- Logical Expressions
    - <, >, <=, >=, ==, !=, !, &&, ||
    - One difference: result of evaluation is of type int. (In Java it is always boolean). When C was created, was no boolean type.
        - 0 – false, 1 – true

- So, Warning: a < i < b is not the same as (a<i) && (i<b)

  a = 5, i = 4, b = 2
  a < i < b equates to (a < i) < b which equates to 0 < 2; the final result is 1.

- 'Short-circuit' Evaluation: Have the same in Java
  - In regards to &&, ||.
  - For example, if in an OR operator, the first part is false, the remaining part of the expression need not be evaluated.
  - So, order of subexpression evaluation for these logical AND/OR operators is defined (this is the exception).
- No boolean types in C89 or earlier.
  - Whenever we see a value that is 0, we treat it as false. For example, in the if statement, rather than in comparisons.
  - If non-zero, treat is as true.
  - In C99, added a boolean type. `_Bool`
    - For example, `_Bool flag;`
    - This type is still an integer type but can only have values 0 and 1.
    - So, flag = 5 assigns 1 to flag (true).
  - `<stdbool.h>` is a header that has some macros.
    - _Bool is assigned to bool. true is assigned to 1 and false is assigned to 0.
    - We could then do: `bool flag; flag = true;`

- Control Structures
  - Selection, Iteration Statements
    - if, switch, for, while, do...while
    - Same syntax as in Java.
    - When used, if the result of the test expression is non-zero this means true, and if zero, it is false. For example: while (1) { } is the same as while (true) { }

Readings: Chapters 4, 5, 6.1-6.3. Next Time: 6.4 – 6.5, 7. Project: Proj. 8 (pp. 96).
Can now work on assignment 4 (two weeks time).
Written exercises will be online.