Projects, Final Exam, and Weights

There are six programming assignments and one written, proctored final exam. The final exam is open notes and open book, but you may not use any electronic devices during the exam. The six programming assignments count for 61% of your grade, the final exam counts for 35% of the grade, and class participation counts for 4% of the grade.

Your grade for the course should reflect what you know and can do as the course ends. I think of your grade as a very brief letter of recommendation to your next instructor or a possible employer. Therefore, I may adjust these fractions if your final exam shows very significant improvement or very significant decline relative to your assignments.

If the final exam shows significant improvement, that might mean you struggled earlier in the term but finally figured it out near the end.

Similarly, if your work on the final exam is significantly weaker than your work on programming assignments, that might mean you forgot most of what you knew during the term or you did not internalize enough of the ideas to draw on them during an exam.

In both cases, I may put more weight on the final exam so the course grade reflects more accurately where you are at the end.

Program in C for Unix/Linux

All projects for this course are programming assignments. You may write and test the code on any system, **but** you must make sure your code is copied to, compiles on, runs on, and is submitted from the course server: cscie28.dce.harvard.edu . We expect you to write in C, the language of Unix system programming. Do not submit solutions in other languages.

What We Look For

Homework assignments are graded on a 100 point scale. Those 100 points are divided into:

Correctness	works correctly
Modularity	file/function decomposition
Efficiency	good use of resources
Clarity	easy to follow
Documentation	comments, Design Docs

Correctness counts for 70 points, the rest for 30 points. Producing a working program is the first step. A working program, then, must be updated, fixed, reused, and read by other people.

If your program works correctly but is poorly designed, you get a C. See the section on *Design and Coding Standards* for more detail.

Letter Grades

The Extension School website states the meaning of letter grades at Harvard. Here are the posted standards for grades of A, B, and C:

A and A-
: Earned by work whose superior quality indicates a full mastery of the subject, and in the case of A, work of extraordinary distinction. There is no grade of A+

B+, B, and B-
: Earned by work that indicates a strong comprehension of the course material, a good command of the skills needed to work with the course materials, and the student's full engagement with the course requirements and activities.

C+, C, and C-
: Earned by work that indicates an adequate and satisfactory comprehension of the course material and the skills needed to work with the course materials, and that indicates that the student has met the basic requirements for completing assigned work and participating in class activities.

I do not grade on a curve. Everyone can get an A, and everyone can flunk. Your success has no effect on the grade of your classmates.

Academic Integrity

The following language is from the DCE website:

> *Plagiarism*
>
> Plagiarism is the theft of someone else's ideas and work. It is the incorporation of facts, ideas, or specific language that are not common knowledge, are taken from another source, and are not properly cited.
>
> Whether a student copies verbatim or simply rephrases the ideas of another without properly acknowledging the source, the theft is the same. A computer program written as part of the student's academic work is, like a paper, expected to be the student's original work and subject to the same

standards of representation. In the preparation of work submitted to meet course, program, or school requirements, whether a draft or a final version of a paper, project, take-home exam, computer program, placement exam, application essay, oral presentation, or other work, students must take great care to distinguish their own ideas and language from information derived from sources. Sources include published and unpublished primary and secondary materials, the Internet, and information and opinions of other people.

Extension School students are responsible for following the standards of proper citation to avoid plagiarism. A useful resource is The Harvard Guide to Using Sources prepared by the Harvard College Writing Program and the Extension School's tips to avoid plagiarism.

Inappropriate Collaboration and Other Assistance

Collaboration on assignments is prohibited unless explicitly permitted by the instructor. When collaboration is permitted, students must acknowledge all collaboration and its extent in all submitted work. Collaboration includes the use of professional or expert editing or writing services, as well as statistical, coding, or other outside assistance. Because it is assumed that work submitted in a course is the student's own unless otherwise permitted, students should be very clear about how they are working with others and what types of assistance, if any, they are receiving. In cases where assistance is approved, the student is expected to specify, upon submission of the assignment, the type and extent of assistance that was received and from whom. The goal of this oversight is to preserve the status of the work as the student's own intellectual product.

The following language is for CSCI-E28:

The work you submit must be your own work. You may base your work on samples from class or examples from texts. We encourage students to discuss ideas, problems, techniques.

Do not show other students your code. Do not look at code written by other students.

Your homework should be all your own work or a combination of your own work and your synthesis and extension of examples. Please state the sources of any piece of code you use, including code from the textbook and class samples.

Do Not Use Generative AI

Course Goals: The goals of CSCI-E28 are to help you understand the Unix/Linux system API and to improve **your** programming and design skills. In the same way that using Google Translate to do assignments for a course in French language and culture prevents students from actually learning French language and culture, using ChatGPT or other generative AI system to produce syntax, algorithms, and problem-solving prevents **you** from actually learning syntax, algorithms, and problem-solving.

In order **to achieve these goals**, we expect students to practice syntax, algorithm design, and problem solving. We expect that all work students submit for this course will be their own. We specifically forbid the use of ChatGPT or any other generative artificial intelligence (AI) tools at all stages of the work process, including preliminary ones. Violations of this policy will be considered academic misconduct. We draw your attention to the fact that different classes at Harvard could implement different AI policies, and it is the student's responsibility to conform to expectations for each course.

Submitting Homework

Homework is due by midnight on Saturday evenings. There is a 10 point penalty for each day late you turn in an assignment. You will submit code and text electronically. Please see the course website for an explanation of submitting your work by computer.

Creating Sample Runs

For most projects we require a sample run of your program. Use the *script* command to capture sample runs of your program. *Script* records everything that appears on the screen and saves it all to a file.

To make a script, type `script`. The computer will print a message and print the shell prompt. Now run your program. Type "exit" at the prompt when you wish to stop recording. Unless you specify some other name, script will save everything in a file called "typescript", so include that file. A sample session is shown below:

```
$ script
Script started, file is typescript
$ cat foo.c
  . . .
$ ./a.out
  . . .
$ exit
Script done, file is typescript
```

Late Days and Catastrophes

Your assignments will lose 10 points per day late. But we know things take longer than planned, and we also know that big problems come up.

If a project takes longer than you expect or if something serious happens, you do not have to ask for an extension. You get four late days and one catastrophe included for free. Here's how it works.

Four Late Days

Across all the assignments, you can use four late days without penalty. You can turn in one assignment four days late, you can turn in four assignments each one day late, or any combination.

Catastrophes

But sometimes your kid gets the flu or your job asks everyone to work an extra twelve hours a day to meet a deadline. In other words, some catastrophe. We allow one catastrophe per term. If you have one, you must submit a good-faith effort for that project (which means at least 60% or the requirements). We then drop that grade. If you do not make a good faith effort, we give you zero for that grade. The late days are not used for the catastrophe.

The only exceptions to this policy are:
(a) We will not drop hw5
(b) You cannot be late for hw6

At the end of the term I compute all combinations of late day deductions and catatrophes and use the combination that produces the highest grade.

Design and Coding Standards

CSCI-E28 is a computer science course that counts toward the degree in software engineering. We want to help you learn and improve:

- Unix systems programming ideas and skills
- Computer science ideas
- Software engineering ideas and skills

Therefore we grade your work with an eye on each of these three areas.

Unix/Linux Systems Programming

Unix systems programming ideas include ideas such as file systems, processes, interprocess communications, concurrency. Unix Systems programming skills include how to traverse a directory tree, how to create a process, how to send messages between running programs, and how to coordinate actions of multiple processes. We will grade you on how well you understand and use these ideas and skills.

Computer Science

Computer science is the field of solving problems by developing and analyzing algorithms machines can perform. The field includes knowing and using algorithms and data structures effectively. We look for effective and efficient algorithms and data structures. These standards apply at the large level such as deciding on a recursive vs iterative solution and at the small level like allocating a temporary string using malloc vs using local a local variable. (Hint: malloc is an inefficient choice)

Design and Engineering

Please follow these five rules for writing clear, readable, maintainable code:

Rule 1: Modular and Layered
Rule 2: Short Functions: 30 lines x 80 cols max
Rule 3: Comments: File, Function, Paragraph
Rule 4: Spaces, Blank Lines, Indenting
Rule 5: Clear, Concise Names

Rule 1: Modular and Layered

The most important rule is that your code be modular and layered. This rule applies to code and to data. Here are the details. Your program will be composed of one or more source files. Each file represents one component of the program. Each file consists of one or more functions, each represents one component of the file.

In our first big example in lecture two, we build a version of the Unix *who* command. This program consists of two files: a main file with logic to read and process user login records and a buffering library file to perform low-level data input from the disk. That main function contained a loop to read and display data. The code to display the data was at a lower layer in its own function, and the code to display the date and time was at a yet lower level also in its own function.

This separation of the solution into functional units ([a] main logic and [b] low-level buffering) is what we mean by modularity. The term *layering* refers to keeping each level in the solution in its own module. The main program knows it will be able to read login records from the disk but does not care about the details of how that gets done. The low-level disk buffering code has one job: read disk data in chunks from the disk and deliver disk records in smaller units to its client. The low-level buffering code does not care what its client does with the data.

No Global Variables: "Modularity and layering" also applies to variables. Each function uses local variables to do its work. Local variables appear when the function is called and vanish when the function ends. A file may have static state variables that are shared by the functions in that file. **Avoid Global Variables**: there is rarely any reason to make data shared by all functions in a program. Global variables undermine modularity and layering. The best way for functions to communicate is through the pass values in, return a value back mechanism.

Design your programs as collections of layers/modules. Implement each module as a separate file. This makes your code more reusable, testable, maintainable.

Rule 2: Short Functions: 30 lines max

The principles of modularity and layering also apply to the contents of each file. A file contains functions. Each function must do one thing and express one layer of abstraction of the solution.

A function must be short enough to be viewed and understood in a single terminal window. Thus: no more than 30 lines tall and no more than 80 columns wide. We grade homework with your code in one window and a grading scorecard in another. Those two windows must fit side by side on one screen.

If a function is too long, it is probably doing too much. That extra work adds complexity and increases the chance of errors. If a function gets too long, split it into smaller functions, each with a specific purpose and a specific level of abstraction. Writing short functions can take a lot of effort. The work is worth it.

Rule 3: Comments: File, Function, Paragraph

Comments are written by the programmer for him/herself to plan and review code and also for the next programmer who has to use and/or maintain the code. Clear writing helps produce clear code.

File Comments A file is a module that implements one component of your solution. The top of each code file must contain a summary of the title, purpose, main features of the module, and brief descriptions of the data structures used by that module.

Say you are a new person on a team and are given a module to debug or extend. You could read the code to try to figure out what the programmer was doing, *or* you could read a nicely written description of the module -- a \`quick start guide´ -- and understand the module quickly and easily. Which one would you prefer to do? Guess which one we, as graders and teachers, prefer.

Function Comments A function is a module within a module. Every function has a specific purpose. Precede each function by a comment that includes:

```
/*  is_logname_valid(const char *s)
 *  purpose: determine if string represents
 *            a valid logname on system
 *     args: string
 *     rets: 1 if true, 0 if not
 *     note: this function ignores case
 */
int is_logname_valid( const char *s )
```

A function is a black box that is passed data, does something, and returns a result. The function comment describes the interface. The code should be short enough and clear enough so the actual algorithm does not get described in the function comments. But, any subtle logic or special handling that is not clear deserves a brief description. Reading code should not be the same as reading a mystery novel. Spell out what is going on.

Internal Comments As just mentioned, in a 30-line function with clear layout (see below) and good naming, the logic is usually pretty easy to follow. Nonetheless, please add brief comments as needed to help the reader see any important or tricky steps.

Rule 4: Spaces, Blank Lines, Indenting

Use Spaces: Donotwritecodethatlookslikethis.

Instead, use space characters to separate words and operators. By doing so, you save the reader the extra work of parsing your code into words and operators. Consider:

```
/* not readable */
   if(x->prev->val<=x->val||p!=x)
/* much better */
   if ( x->prev->val <= x->val  || p != x )
```

Got it?

Use Blank Lines: Just as blank spaces between words and operators increases readability horizontally, blank lines between paragraphs increases readabililty vertically. Imagine if the document you are now reading were written without any blank lines between paragraphs.

Indent 8 spaces (4 if you insist) All conditional blocks, function bodies, loop bodies, etc, must be indented relative to the control line. Please use 8 char tabs (or 4 if you object to 8). Never fewer. Use tabs or spaces as you please but be consistent.

Rule 5: Clear, Concise Names

Function names and variable names must be explanatory but not verbose. For example:

```
num_users
```

is fine but

```
number_of_users_logged_in
```

is too wordy.