What is CSCI-E26?

CSCI-E26 is a course in C and Unix/Linux programming with a focus on using C and Unix/Linux to create interactive pages for the World Wide Web. The course covers in detail almost of the the C language, introduces the major tools and ideas of Unix programming, and shows how to use HTML forms to connect to remote databases and services.

Who is prepared for it?

We assume you know how to write computer programs. In particular, we assume you have written programs in a structured language, that you know about editors, understand the ideas of variables, loops, arrays, functions, and some data structures. We expect you to have taken a course in data structures that include using linked lists. If you have not written complicated programs that use these ideas in some language, you will be overtaxed by the course.

Administative Details

Lectures Wednesdays, 7:40-9:40 PM ET using Zoom. short break. Covers ideas, sample programs. Be prepared to take notes. All sample programs used in class will be stored on line, so you can retrieve them and examine and/or print them later.

Homework Several assignments, due on Sunday evenings at midnight. See course outline. Must hand in listing and sample run. Must run on the course machine: cscie26.dce.harvard.edu. Can be developed on any machine. For details about the homework, see the Assignments page on the course web site (cscie26.dce.harvard.edu/~dce-lib113).

Exams One midterm, one final.

Grading The weighting is roughly: homework 45%, participation 5%, midterm 20%, final 30%

Sections One hour each week at time to be determined.

Office Hours Online with Zoom, times to be arranged

Questions/ Discussion Use Ed Discussion on Canvas to ask questions of staff and other students and to start class discussions.

Info Sheets Need name, address, programming experience, and section requests. Complete the online form at https://cscie26.dce.harvard.edu/~dce-lib113/infoform/ .

Help CSCI-E26 provides several forms of support to help you learn the material and succeed with the projects:

- Students are encouraged to ask questions in class
- I call on students in class<
- Weekly sections discuss main ideas and homework strategies
- Weekend workshops (with Zoom) provide hours of time to discuss ideas, meet other students, work in company of others
- Ed Discussion forum for posting questions to the class or directly to teaching staff
- One-on-one meetings by request
- After-class zoom discussions

Texts The required texts are *C Programming, A Modern Approach* by King (1st or 2nd edition), *Your Unix: The Ultimate Guide, 2nd ed or 3rd ed.* by Das.
The Harvard Coop has them. Online book sellers have them. They are for reference and additional examples. The suggested reading does not exactly follow lecture, but is pretty close. The *C Programming* book explains the language, the Unix book explains how to use Unix.

Facilities The course machine is cscie26.dce.harvard.edu. Connect using ssh over the Harvard VPN. Instructions for the VPN can be found here: https://harvard.service-now.com/ithelp?id=kb_article&sys_id=f90a73f6dba21c5c60c0d9fcd39619ea

Accounts You will have an account on the E26 server. Your username on the E26 server is your Harvard NetID. You can find your NetID at https://key.harvard.edu/manage-account.

Accounts will be available one week before classes start. You need a *Harvard Key* to set up your account. Claim your *Harvard Key* at https://key.harvard.edu/ . For details about claiming your key, visit: https://extension.harvard.edu/for-students/support-and-services/computer-and-e-mail-services/

Web Site https://cscie26.dce.harvard.edu/~dce-lib113/

Accessibility

The Division of Continuing Education (DCE) is committed to providing an accessible academic community. The Accessibility Services Office (ASO) is responsible for providing accommodations to students with disabilities. Students must request accommodations or adjustments through the ASO. Instructors cannot grant accommodation requests without prior ASO approval. It is imperative to be in touch with the ASO as soon as possible to avoid delays in the provision of accommodation.

DCE takes student privacy seriously. Any medical documentation should be provided directly to the ASO if a substantial accommodation is required. If you miss class due to a short-term illness, notify your instructor and/or TA but do not include a doctor's note. Course staff will not request, accept, or review doctor's notes or other medical documentation. Please visit https://www.extension.harvard.edu/resources-policies/accessibility-services-office-aso for more information, or contact accessibility@extension.harvard.edu.

Academic Integrity

If you do not know how to start a project, feel overwhelmed, need help with some bugs, we offer a lot of support. There is no need to compromise academic integrity.

You are responsible for understanding Harvard Extension School policies on academic integrity and how to use sources responsibly. Violations of academic integrity are taken very seriously.

Not knowing the rules, misunderstanding the rules, running out of time, submitting the wrong draft, or being overwhelmed with multiple demands are not acceptable excuses. There are no excuses for failure to uphold academic integrity.

Review important information on academic integrity and student responsibilities here: https://extension.harvard.edu/for-students/student-policies-conduct/academic-integrity ; for more on academic citation rules, visit Using Sources Effectively and Responsibly (https://extension.harvard.edu/for-students/support-and-services/using-sources-effectively-and-responsibly) and review the Harvard Guide to Using Sources (https://usingsources.fas.harvard.edu).

Details for Academic Conduct

Unless otherwise stated, all work submitted as part of this course is expected t o be your own.

You may discuss the main ideas of a given assignment with other students (provid ed that you acknowledge doing so in your solution), but you must write the actua l solutions by yourself. This includes both programming assignments and other ty pes of problems that we may assign.

Prohibited behaviors include:

- Copying all or part of another person's work, even if you subsequently modify it
- Viewing all or part of another student's work
- Showing all or part of your work to another student
- Consulting solutions from past semesters,
  or those found in books or on the Web

If we believe that a student is guilty of academic dishonesty, we will refer the matter to the appropriate administrative committee. Penalties for this type of behavior are typically severe.

Generative AI

Course Goals: CSCI-E26 is a programming course using C and Unix. The goal of the course is to help you learn C and Unix programming 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 Chat-GPT 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.

Course Material	Publishing or Distributing Course Materials: Students may not post, publish, sell, or otherwise publicly distribute course materials without the written permission of the course instructor. Such materials include, but are not limited to, the following: lecture notes, lecture slides, video, or audio recordings, assignments, problem sets, examinations, other students' work, and answer keys. Students who sell, post, publish, or distribute course materials without written permission, whether for the purposes of soliciting answers or otherwise, may be subject to disciplinary action, up to and including requirement to withdraw. Further, students may not make video or audio recordings of class sessions for their own use without written permission of the instructor.
Attendance/ Participation	Students are encouraged to attend class during the live presentation and to participate by asking and answering questions. Participation counts for 5% of the grade and also includes participation in online sections and office hours. Students who cannot attend class may participate in sections, office hours, and/or discussion site.
Credit/Work	Graduate-credit students will submit additional design/planning documentation for class projects.

What is the Point of this Course?

This is a course in Unix/Linux programming. Unix is an operating system, but it is more than just a control program for computers; it is a complete programming environment based on the idea of software tools. The Unix programming model is one of building complex, powerful solutions by combining simple, special-purpose tools. These tools are data manipulation programs. Every Unix system comes with a wealth of these tools. Tools are usually written in C. Combinations of tools are often written in a scripting language, such as the Unix shell, *sh* .

Unix programming, then, consists of **(a)** designing and writing tools in C and **(b)** combining them using sh. Csci-e26 teaches C programming and sh programming.

Where do web interfaces fit into this? To make your combination of tools available from web pages, you need to learn how to connect web pages to Unix programs. The *Common Gateway Interface* is the method web pages use to transfer data from web page to a server and back. By learning this third skill, you will be able to construct Unix tools, combine them into Unix programs, and use web pages for user input and output.

Each of these three skills is useful on its own. C is a great language; its syntax is the basis of C++, JavaScript, Perl, awk, Java. Unix is written in C as are most of the software tools. Shell scripting is an essential skill for Unix administration and for power users. CGI programming may be done in any language, not just C/Unix. The principles we cover apply to any network and programming platform. These principles are the basis of all web-based applications.

DATE	CLASS	READING	SECTION	HOMEWORK
Sep 3	Overview Unix/C and the web Sample Program	D: Ch 1,2,3,5 or 6 K:Ch 2,3,4	Using Unix Files and Dirs Editors	Assignment 0 due Sep 7
Sep 10	The structure of C programs Functions and filters Arrays and Strings	D:Ch 4,7,10 K:Ch 5,6,7,9	Functions and strings	
Sep 17	Arrays and Strings More Loops Generating HTML	K:Ch 8, 10, K:Sect 13.1-13.5	Arrays and memory debugging	Assignment 1 Short Ones due Sep 21
Sep 24	Interfaces: HTML forms and Scripts structs, arrays, functions	See web page	Shell scripts Forms	
Oct 1	Pointers Strings Functions	K:Ch 11,12 K:Sect 13.6	Using pointers	Assignment 2 Table Converter due Oct 5
Oct 8	Structs Dynamic Memory Allocation Linked Lists	K: Sect 16.1-16.2 K: Sect 17.1-17.6	Linked lists Pointer Bugs	Assignment 3 Paper Pointers due Oct 12
Oct 15	More Pointers Multi-File Programs	K:Ch 14, 15	Pointer roundup Using make	
Oct 22	File I/O Command Line Args	K:Ch 18, 22 K:Sect 13.7	Test Review	Assignment 4 Word Freq due Oct 26
Oct 29	Midterm Exam			
Nov 5	Shell Programming 1 Scripts and args	D:Ch 13	Focus on Files and argv[]	
Nov 12	Shell Programming 2 Wildcards and Loops	see web site	functions and lists	Assignment 5 Formletter due Nov 16
Nov 26	No Class Thanksgiving Eve	See Web Site	loops and quotes	
Nov 19	Shell Programming 3 Web Interfaces	See Web Site	loops and quotes	Assignment 6 Shell Scripts due Nov 30
Dec 3	Web Programming I: Distributed Processing	See Web Site	How a web server works	
Dec 10	Web Programming II: Distributed Data Sources	See Web Site	Final Review	Assignment 7 due **SAT** Dec 13
Wed Dec 17	Final Exam			
Jan 28	First lecture of CSCI-E28 *Unix/Linux Systems Programming*			

Last update: 2025-07-25

ACADEMIC HONESTY

The work you submit must be your own work. You may build your code on samples from class or examples from texts, and we encourage students to discuss problems and techniques. Your homework should be all your own work or a combination of your own work and your synthesis and extension of class examples. You must cite any sources.

You are responsible for understanding Harvard Extension School policies on academic integrity (www.extension.harvard.edu/resources-policies/student-conduct/academic-integrity) and how to use sources responsibly. Not knowing the rules, misunderstanding the rules, running out of time, submitting the wrong draft, or being overwhelmed with multiple demands are not acceptable excuses. There are no excuses for failure to uphold academic integrity. To support your learning about academic citation rules, please visit the Harvard Extension School Tips to Avoid Plagiarism (www.extension.harvard.edu/resources-policies/resources/tips-avoid-plagiarism), where you'll find links to the Harvard Guide to Using Sources and two free online 15-minute tutorials to test your knowledge of academic citation policy. The tutorials are anonymous open-learning tools.

GRADING

Homework assignments are graded on a 100 point scale. The 100 points are divided between Function (70 points) and Design (30 points). In software engineering, getting a program that works is only part of the problem. The rest of the problem involves updating, correcting, and reusing the code.

Thus, if your program works right but is designed poorly, you get a C. Design is divided into three categories each worth 10 points: Documentation, Modularity, and Clarity. *Documentation* refers to commenting; files, functions, variables, and chunks of code should be documented. *Modularity* refers to how the program is broken into separate, well-defined, and insulated units. *Clarity* refers to the physical presentation and logical design of the program. Have you ever read a book or essay that expressed an idea or technique so clearly that after just one reading, you saw exactly what the writer was getting at? Have you ever read a book or essay that, even after several readings still confused you? Some code is really clear, and some code is hard to figure out, hard to maintain, hard to modify. Strive for clarity; you'll be graded on it.

A detailed description of what we want will be distributed

TURNING HOMEWORK IN

Due on Saturday at 11:59PM

Homework is due by 11:59pm on Saturday. There is a 10 point penalty for each day late. You must turn in both a listing of all your code and sample runs of your program (how to do this is described below).

Electronic Version

We use an computerized system for submitting and returning homework. You do not have to print any paper for most of the assignments. Details for using the system will be provided with the assignment.

GENERATING SAMPLE RUNS

Use the `script` command to generate the sample runs you will hand in. When you run `script`, everything that appears on your terminal screen is saved in a file. To make a script, type in `script`. The computer will print a message and give you a new, regular prompt. Now list your program (using cat) and run it. 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 print 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
$
```

You can specify a different name for the script file by typing the command `script` *`filename`*. **Note:** Every time you run script the script file will be overwritten.

This is a short assignment. You will submit one C program, one shell script and one pipeline. You will also submit three self-assessment programs.

1. Setup your account and Harvard VPN following the steps at

   `https://cscie26.dce.harvard.edu/~dce-lib113/news/Starting.html`

2. Connect to cscie26.dce.harvard.edu by using one of these:

   Mac OSX users -- open a terminal (Applications:Utilities) then type

   `ssh` *yourNetID*`@cscie26.dce.harvard.edu`.

   Linux/BSD users -- open a terminal then type

   `ssh` *yourNetID*`@cscie26.dce.harvard.edu`.

   MS-Windows users -- open a cmd window and type

   `ssh` *yourNetID*`@cscie26.dce.harvard.edu`.

   MS-Windows users -- download PuTTY (a free web download) and connect to cscie26.dce.harvard.edu using ssh.

3. Login into your account

4. **PART ZERO** Do at least three of the exercises on the self-assessment problem sheet. You must do at least one of the last two problems. Write the programs in any language you like. If you find these difficult, then write to us right away to discuss if you are prepared for E26.

5. Make a new directory for this project and change into that directory by typing:

   ```
   mkdir 26-hw0
   cd 26-hw0
   ```

6. **PART ONE**

   a. Type the following program (from the C Programming book) into a file called first_try.c . This program is in the chapter on Arrays and is in the section with the heading "Reversing a Series of Numbers". Use vi or emacs, and try to use as many features of the editor as you can to correct any typos you make.

```
#include <stdio.h>
/*
 * reverse a list of numbers
 */
#define N 10

int main()
{
        float   a[N];
        int     i;
        printf("Enter %d numbers: ", N);
        for ( i = 0 ; i < N ; i++ ){
                scanf("%f", &a[i]);
        }
        printf("Here are your numbers in reverse order\n");
        for( i = N-1 ; i >= 0 ; i-- ){
                printf(" %f", a[i]);
        }
        putchar('\n');
        return 0;
}
```

b. Compile the program by typing

```
gcc -Wall -Wextra first_try.c
```

to the shell.

c. If all is well, the compiler will produce a file called **a.out** which you can run simply by typing

```
./a.out
```

to the shell. If you get syntax errors from the compiler, you made made a typo. Fix it in an editor, then try again. If there are a lot of syntax errors, fix the first one and recompile. Some errors can confuse the compiler enough that it thinks the rest of your code is all wrong.

d. When you run a.out you will be prompted for ten numbers. Type in some numbers separated by white space (spaces, tabs, or newlines). You can try typing the numbers on separate lines, or on one line. Try putting in negative numbers or words. What does the program do?

e. Then rename this file something more meaningful, say **reverse** by

```
mv a.out reverse
```

to the shell. You can run the renamed program by typing:

```
./reverse
```

f. Try to understand the key elements of the program: how scanf does the inputting and how printf does the outputting. Try to get a feel for the look of a C program.

7. **PART TWO** Experiment with Unix tools to analyze the MBTA Commuter Rail schedule data file by doing:

```
ln -s ~dce-lib113/sched.errs sched
```

This version of the file contains several errors. These are not factual errors such as incorrect arrival time. Instead they are data entry errors such as incorrect tag names or bad data format.

See how many errors you can find in this file. Create a plain text file called *errors* that lists all the errors you find. ☞ **Also** in the *errors* files, describe what tools, procedures, and/or commands you used to find the errors.

8. **PART THREE**

Write a shell script called **busiest-hour** that contains one pipeline to find which hour of the day has the most train stop events on a given day: m-f, sa, or su. The script should take an argument of m-f, sa, or su and print out the hour of the day which contains the largest number of records in the sched file and the number of records for that hour.

9. **SUBMIT YOUR WORK** To submit your work, login to your e26 server account, change into the 26-hw0 directory, then type

```
~dce-lib113/handin hw0
```

CSCI-E26 Course Outline Introduction

Csci-e26 teaches C and Unix programming. C and Unix are used for all sorts of programming - scientific, database, financial, text processing, operating systems. In this course, we shall focus on CGI programming as an on-going example.

The web browser provides a user interface. The C and Unix programs serve as engines to process, store, and retrieve data. We shall see how to build these engines in C and Unix, and we shall see how to use C and Unix programs to connect these engines to web pages. Class 1. The Big Idea

Computer programs have not really changed much in decades. The sequence of events is always something like: 1. The program asks user for input 2. The user types in something 3. The program receives the input from the user 4. The program does some processing, maybe looking up
information in external files, maybe storing data
somewhere, maybe doing some calculations. 5. The program sends results back to the user Everything from a hand-held calculator or video game to a weather analysis program works pretty much along these lines. The basic operation of programs does not change, but the languages and tools for building the programs and the devices used for input and output do change. Once, paper tape and punch cards served as input and output and the programming was done in FORTRAN and COBOL. Later, teletypes were used for input and output, and new languages appeared - BASIC, Algol, Lisp, APL. Later yet, video display tubes with keyboards appeared and even more languages appeared. Video terminals that connected users to central computers began to be replaced by personal computers. On a personal computer, entering user input, processing data, and displaying results were all done on one machine.

Now the world wide web has appeared. The web revives the model of separating the user interface from the data processing system. The web pages are fancier than traditional video terminal screens, but they serve exactly the same purpose. Is this a step backwards? Not really. There is an important difference between the video terminals and the web pages. Video terminals provide access to a single machine using a cable. Web pages provide access to any number of machines using network connections.

With the web, a computer user can send data to computers anywhere on the network and receive the results back. The web has made popular the concept of a *software engine*. People speak of the search engines they like, or engines that generate driving directions between any two addresses in the country. The web has made it possible to think of software the way people think of automobiles - the sedan model with the V8 engine, or the minivan model with the V8 engine, or the minivan model with the v6 engine.

What does this have to do with C and Unix? Simple. C is a language used to build software components - little motors that perform specific tasks. Unix is a programming language designed to combine software components into programs. Some people have complained that C and Unix programs have unfriendly user interfaces. That is like saying that a Chevy straight 6 motor has a crummy dashboard. Unix makes it easy to add any user interface to a C program, including a web interface. Just as the straight 6 engine doesn't care what color the steering wheel is, C and Unix programs don't care what the user interface looks like.

What is this course about, then?

This course will teach you how to program in the C programming language. We shall use C to build software components - special purpose programs with terse, unfriendly, command-line interfaces. The course will teach you how to use the Unix programming language to combine components to create useful applications with friendly user interfaces. Finally, the course will teach you how to create and attach web pages as user interfaces to components and applications.

The C and Unix programming parts of the course are useful even if you never use the web as a user interface. The web interface part is useful even if you decide to program in some other language or operating system.

Unix: Operating System or Programming Language?

In the preceding paragraphs, I have been speaking about the Unix *programming language*, but most people refer to Unix as an *operating system*. What gives? In this first class, we shall see that Unix is an operating system and that Unix is also a programming language.

An Extended Project

We discuss the plans for an extended project: a web site and tools for a database of MBTA commuter rail schedules. (Note for those not from the Boston area: The MBTA is the metropolitan Boston Transit Authority - the agency that runs public transportation)

The web site will include functions for searching, tabulating, and mail-merge. We sketch the general architecture of the system. We shall build the pieces in some of the homework assignments. Class 2. Software Tools

The web offers a user interface. Shell scripts act as dispatchers - collecting requests from the browser and invoking tools to do the work. Software tools do the work.

Consider the paper and pencil activity of registering for a course or two at the Extension School. The paper applica-

tion is the user interface; the applicant puts information into data fields on the form. The form is submitted. Then, at 51 Brattle Street someone opens the envelope and sends the contents to different departments: registrar, computer services, financial services, student records...

We start with software components. A Unix software tool is a simple machine with input, output, storage, and code. The input can be attached to a file or a process. The output can be attached to a file or a process. The tool accepts command line arguments that affect its behavior.

What goes on *inside* of one of these tools? We look at the structure of a C program - functions, variables, input/output, control flow. Unix and the web work with text, so we focus on programs that process characters and strings.

Strings are simply arrays of characters. We study arrays.

All transactions on the Web are text-based. Most Unix programs operate on textual data. We write some programs to process the train schedule database: one to capitalize names, one to convert semicolons to tabs. We add these hand-crafted tools to our set, and we incorporate them into shell scripts. Class 3. String Processing: Lists of variable=value

A web browser accepts data and sends it across the 'net . The fields and their values are encoded as a set of var=val pairs. The shell uses var=val to to store session information. Our train schedule db db uses var=val to store info. This is a popular format. Let's see how to write programs that work with this model of information storage.

We look at data formatting programs.

We write code to parse a line like

```
TR=1806;dir=i;day=sa;TI=12:10;stn=hyde park;Line=attleboro
```

into separate strings and then examine and output those strings.

String operations are so central to Unix programming that the system comes with a lot of functions to assign, copy, compare strings in all sorts of ways.

Once we collect our data formatting tools, data analysis tools, and data cleaning tools, we have the pieces we need to create scripts to analyze, clean, format, process all sorts of textual data. Class 4. Building Interfaces: Shell Scripts and Web Pages

We have seen how to construct software engines to perform specific operations. We can use these components directly, typing in data or sending it command line arguments. It is now time to create user interfaces to these components.

We start with shell scripts that prompt the user for information and then passes that information to the engines. We use shell variables to store user responses. Sometimes we use variables to store output from the engines.

In the second half of the class, we step back one more level - across the Internet. We create a simple web page that uses a *form* to accept user input. When the user 'submits' the form, all that input is delivered to a program on the Unix machine. That program then passes that input to the engine. The output from the engine is then sent back to the user over the Internet.

The data input part of the program runs on the user's computer, while the data processing part runs on the web server. The questions we look at are: 1. How does one create a form? 2. How does the input get from the form to the server? 3. How does the output get from the server to the user? We are introduced to a software tool (an engine) that converts the user input values into shell variables. With it, we have a continuous path from browser screen to C program and back.

We work through several examples to get accustomed to this three-tier architecture. Class 5. Strings and Pointers

Strings are a central data type in C/Unix and Web programming. We look at them in more detail. How are they stored? How does one process them? How do they get passed to functions? We meet pointers. We use pointers to process strings. The idea is not too tricky, but the syntax takes some getting used to. We do lots of examples; pictures are provided. Class 6. Structs and Linked Lists

One of the functions provided by our web site is to tabulate data. For example, we can ask how many trains stop daily at a given station, or for a given station, we could ask the times of the trains that stop there or the frequency of trains per hour.

We map out an abstract filing system for this project. We then look at three implementations: parallel arrays, an array of structs, a linked list. Pointers figure prominently in our work. Class 7. Managing Complexity: Multi-file Programs, Storage Classes

As our programs start to grow, we need a way to keep the code simple and maintainable. C programs can be written

in several files and linked together. We see how it's done.

Once you split a program into separate files, you need to think about how that division affects variables. We see how files can have local variables just as functions can have local variables. Class 8. File I/O and Command Line Arguments

Our FirstYear data access system processes data from a file. Many programs work with files of data. How can a C program read and write data from a file? We see how to open a file, get characters and strings from it, put characters and strings into a file, and how to close a file.

The 'user interface' for most Unix programs is the command line. It allows the user to specify file names, parameters, and options. To complete our study of writing software engines, we see how to process command line arguments.

We then go back to a web page and show how check boxes or radio buttons can be used to invoke command line options on the C program. Class 9. Focus on Shell Scripts: The Middle Tier

Having studied C in detail, we now turn our attention to the middle level - shell scripts. We have been using shell scripts to transfer data between the web browser and the software tools.

The shell is a powerful programming language in its own right, with variables, if..then, loops, functions. For some projects, it is a better tool than C.

We look at several examples to see how to use write shell scripts that do interesting, and sometimes complicated, things. Class 10. Shell Scripts Part II: Loops

We continue to learn about using the shell as a programming language. We focus on file and directory processing, using wildcards and loops. Class 11. Shell Scripts Part III: CGI in sh

The shell is a programming language, so one can use it as the processing part of a CGI program. It is particularly useful for indexing directories and building dynamic html. Class 12. Web Programming: Design Decisions

We also look at alternatives to C/Unix CGI programming. A web-based program spans three levels: web page, request handler, software tools. Where on these three levels do you store data? Where do you do processing? What are the options? What are the trade-offs?

Now that we have studied the details of all three levels, we consider design decisions for building complete applications. Class 13. Unix Text Tools: Regular Expressions

We have worked with data in text files, but there are other types and sources of data (images, sound), other sources (program output, devices, web services), and other processing tools (netpbm, sox, regular expressions). Today we explore some of these other sources, types, and associated Unix tools.

Unix: A Programming Language Part 1: Train Schedules Introduction

Unix is more than a system for running applications and servers. It is a programming language. What does that term mean? Consider popular programming languages like Java and C++. These languages provide three things: (a) A set of functions and operations you can apply to data, (b) a syntax for combining those functions and operations into programs, and (c) a way of executing those programs.

Unix provides these three things, too. Unix provides a large number of tools that operate on data, Unix provides a syntax (the shell scripting language) for combining those tools into programs, and Unix provides a way to execute those scripts.

Unix programming consists of writing scripts that invoke tools to process data. Sometimes, though, there is no tool for a particular job. In that case, you can write a new tool and add the tool to the system. For Unix, tools are usually written in C. Programs in C are fast, portable, and compact.

In this course, we shall see how to use existing tools to build Unix applications, how to write new tools in C, and how to make these applications accessible via web pages. We begin with an example. The Problem: Processing Train Schedule Data

The MBTA commuter rail system runs lots of trains each day on several lines. The MBTA web site provides access to information about those trains and lines. Travelers can view maps and schedules on the web site. Travelers can also ask the system how to get from one place to another. The computer checks the schedules and reports which trains to take and where to change trains if necessary. How does that all work?

Imagine the MBTA had asked you to build their system. Consider the following plausible scenerio. Someone in the MBTA transcribed all printed train schedules into a spreadsheet and then saved the spreadsheet as a big text file. This person read down each column on each schedule recording every stop every of every train. Here is the format:
TR=070;dir=i;day=m-f;TI=5:40;stn=greenbush;Line=greenbush TR=070;dir=i;day=m-f;TI=5:47;stn=north scituate;Line=greenbush TR=070;dir=i;day=m-f;TI=5:54;stn=cohasset;Line=greenbush

through $STATION" grep "stn=$STATION" sched | grep "dir=$DIR" The output of this script is not in a user-friendly format. We could make this script part of a pipeline, but there is a problem. The script prompts for a station name and a direction. A cleaner way to pass those two values to the script is to pass them as command line arguments. That is, we can modify the program so we can type: train-times wakefield o where the station is the first argument and the direction is the second argument. A command-line argument version of the script is: #!/bin/sh # # train-times-args # purpose: list train times for a station # usage: train-times-args stationname direction # where: direction is "i" or "o" # STATION=$1 DIR=$2 grep "stn=$STATION" sched | grep "dir=$DIR" The difference between these two versions of the script is important. The first script interacts with the user to get the values it needs. The second script gets the values from the command line.

Designing scripts that accept arguments on the command line makes them into tools that can be included in pipelines and in other scripts. For example: train-times-args salem o | cut -d";" -f1,3,4 But, sooner or later we need to add a nice user interface, don't we? Web Pages as Nice User Interface

The web lets you put nice user interfaces on your shell scripts. You need a way to get the values for the user and a way to read those values and pass them to the shell script that does the work.

Therefore, putting a shell script on the web requires only two things: (a) An HTML form, and (b) a script to process the form. Here is an example of how to put the train-times-args script on the web:

The HTML Form: train-times.html <html> <body> Find train times for a station <p> <form action="train-times.cgi" method="get">

```
Station: <input type="text" size="25" name="station"><br>
Direction (i or o): <input type="text" size="1" name="dir"><br>
<input type="submit"> </form>
```

The Form Processing Script: train-times.cgi #!/bin/sh # processing script for train-times.html

```
eval $(./qryparse)
echo "Content-type: text/plain"
echo ""

echo "Train times for $station direction $dir"
./train-times-args "$station" "$dir" Three Languages - One Big Idea This course teaches three
```

languages, one for each level in the picture The user interface level is based on HTML web pages using HTML forms to collect user input. We shall learn how to design these pages and forms. Shell scripts combine tools to process data. Scripts can be used directly from the command line, and scripts can receive data from web forms and invoke scripts and tools to do the work. The shell is a programming language with variables, control flow, functions, and many other features. We shall learn how to program in this language. Most Unix tools are written in C. We shall learn how to program in C and how to design programs that can be used as software tools.