CSCI-E26: Class 2 -- Writing Software Tools in C

0. Outline and Intro

1. Quick review of the big picture -- trainsched.html
2. Software tool - the basic model
3. Some simple tools - semi2tab, capitalize
4. Putting the new tools to work
5. C programs - structure, storage, control
6. Programming with strings

Last week we started with a simple story: A traveler at the Salem station wants to know the times of trains going into Boston. The information is all stored in a simple text file. Here is another simple story: A traveler at the Salem station plans to take train 181 and wants to know the schedule for that train. How can we connect the user to the data? We fill in the pieces in this picture:

1. Reviewing the Big Picture

   We use simple tools, combined into scripts and pipelines, to build a program that finds and displays a train schedule. Then, using HTML forms and a connector program, we connect Internet users to our program.

   This week we begin writing software tools. Most Unix tools are written in C. Programs written in C are compact, portable, and run quickly and efficiently. C is a remarkably stable language. Programs written thirty years ago can still be compiled and run without changes.

   Let us demonstrate the parts of this model by writing a web page to display the schedule for a train. At left is a script called trainsched that takes as an argument the train number.

   The tools here are grep, cut, and sort. We combine these tools in a pipeline in a shell script to produce a program. We can use this program from the command line.

   !/bin/sh
   # trainsched
   # usage: trainsched train#
   # action: finds all entries for that TR and
   # shows time and stn cols
   #
   if test $# != 1
   then
     echo "usage: trainsched trainnum"
     exit 1
   fi
   grep "TR=$1" sched | cut -d";" -f4,5 | sort -t= -k2n

   We put this program on the web by building two files: an HTML form and a connector script.
The HTML form:

```html
<!DOCTYPE html>
<html><head><title>Show Train Schedule</title></head>
<body>
<form action="trainsched.cgi">
  Train number: <input type="text" name="trainnum" size="5" maxlength="5">
  <input type="submit" value="Show schedule">
</form>
```

finally, The Connector Script:

```sh
#!/bin/sh
# trainsched.cgi
# connector between trainsched.html and trainsched
#
eval $(./qryparse)
echo "Content-Type: text/plain"
echo ""
./trainsched $trainnum
```

We can use the trainsched program from a web browser anywhere on the Internet. Output is accurate, but it
looks crummy. If we want to process the output, we just filter it through some software tools.

2. Software Tools - The Unix Philosophy -- Components and Connections

The essential ideas of The Unix Philosophy (search it) are simple components and connections. These simple
components are called software tools. Most software tools read in data, processes the data, then output a result. A tool looks like:

```
input ---+--- process ---+--- output
```

Input comes from a file named on the command line or from standard input. Standard input is the keyboard
by default but may be attached to the output of a program using the pipe symbol.

A key feature of this model is that software tools can be snapped together into pipelines:

```
input ---+--- process1 ---+--- process2 ---+--- output
```

Writing a software tool is pretty easy. Three parts of the picture refer to the three parts of the program: (1)
input data (2) process data, (3) output data. Let’s write some tools to process this file of train schedule
information. Doing so will introduce us to some basic elements of C programs.

3. Some Simple Tools

We shall write some tools to process the output of the trainsched script. Two things to fix are: (1) change
the semicolons into tabs so the columns are clearly separated, and (2) capitalize the station names. We shall
write these tools in the C programming language.

Facts about C

C is a compiled language. You write the program using a text editor, then you compile the program into
machine language -- the native code for the computer. You then run the compiled program, sometimes
called the ‘executable’ or the ‘binary’. The three steps of creating and running a C program are:
STEP COMMAND
write editor program.c
compile cc -Wall program.c -o program
run ./program

The fourth step, not shown, is to debug. Several tools are available for this step.

semi2tab1.c This C program reads in data, character by character, and writes out data replacing each semicolon with the tab character. Features of this program are:
   a. basic outline of program: directives, comments, function
   b. basic outline of function: name, arglist, variables, code
   c. the while loop: a condition, a set of braces, a break to exit
   d. input/output: getchar and putchar
   e. conditional execution: the if statement

semi2tab2.c This second version shows the more typical way of coding this loop. The getchar and test may be all included in the while condition. This is how you’ll see it written.

We can use this program as part of a pipeline with our other tools and scripts:

   ./trainsched 181 | ./semi2tab
   ./trainsched 181 | ./semi2tab | expand -20

Those tags are a nuisance. For homework, you’ll modify semi2tab to remove the tags, also. What about capitalization. That would make the output look much nicer.

capitalize.c This program (1) inputs data, (2) capitalizes letters following an equals sign or a space, and (3) outputs data. A later version could be more general. We add that.

We add this tool to our gradually growing toolbox.

   ./trainsched 181 | ./semi2tab | expand -20 | ./capitalize
   ./trainsched 181 | ./capitalize | ./semi2tab

4. Putting the new tools to work

We wrote semi2tab and capitalize to make the web report look better. We can use them now. Do we put the new tools into trainsched or into the connector script? This is a design question. How do you decide?

5. Tools for Analysis

We can also write tools to analyze data, useful for cleaning. Each line in sched should have five semicolons. What if a line has the wrong number of fields? That will produce bad output. We could write a program to search for lines that do not have five semicolons or we could write a tool to count semicolons. The latter is more useful. Later, we learn to print the entire line as well as the count and the line number.

6. C Programs - structure, storage, control

If we are to write more sophisticated tools, we need to study the basic structure of C programs. We’ll start with some simple C programs and add features.

The important topics we need to learn about are:
   a. The structure of a C program
   b. The structure of a C function
   c. Types of data storage - int, char, float, long, double, unsigned, short
   d. Control structures - if, while, for, switch
   e. Strings - arrays of chars

The programs we shall study to learn about these topics are:
hello1.c A program that prints a message. This program shows the main sections: directives, header comments, main function. A function has a type, a name, an arglist, and a body. The body contains variable definitions and code, in that order.

hello2.c We modify the program so it prints the message a number of times. The main function here contains two variables and a loop. The printf function prints text with data values inserted into spots marked with a leading %. The for loop is just like the for loop in Java and Perl.

hello3.c We modify the program so it asks for the message to print and the number of repetitions. Here we show how to use string variables in C. A string is an array of characters. The size of the array has to be specified when the string is defined. The fgets function reads in a string. The %s format prints out a string in printf. The scanf function can read in a numerical value. But it does not handle non-numeric input well.

hello4.c We change the program so the repetition is done in a separate function. We introduce a function to perform a specific task. We pass arguments to this function. We also show how to use the #define directive to create a symbolic value. This is NOT a variable, it is a symbol replaced during compilation. Notice how we declare the function before we use it.

hello5.c We enter a negative number and realize we have to add more code. Do we exit or loop back.

hello6.c We enter a word and realize we need to learn to analyze strings. We solve the scanf problem by reading input as a string and then analyzing the string char by char to see if all the chars in it are digits. This shows how to process strings and how to write functions that return boolean values.

hello7.c We now rewrite the program as a software tool accepting arguments on the command line rather than prompt the user for them. This version does not enforce a legal numerical string for the count argument.

7. More Software Tools

We return to the task of writing software tools to process the train schedule datafile. We wrote tools to format the data so our reports would be easier to read.

And we wrote count_semis1.c to analyze the dataset to see if any records were bad. We now know enough to improve count_semis to print the entire line along with the count and the line number. We write count_semis2.c. And we grep the output to find lines that do not show 5 semicolons.

```
./count_semis2 < sched | grep -v "5 "
```

We can then go back to the original file and correct those lines.

The logic of this program is similar to the character processing programs we wrote earlier. We (a) Read in a line, (b) do something with it, (c) print out a result. In this case, though, we need to store an entire line not just a single character.

What other kinds of errors are there?

- a. Missing data - fields are there, but are blank
- b. Wrong order - we assume fixed positions, but that might be wrong
- c. Data errors - time format is wrong, incorrect station or line

For all these other errors, we can use logic similar to count_semis2.c