0. Introduction

Last week we continued to extend our HTML and shell skills as we improved our train schedule web site. On the C end of the course, we looked at arrays in general and strings in particular. We saw how to write programs that worked with arrays of strings.

This week we continue to improve our website and extend our C skills.

The outline for tonight’s class is:

1. Quick review of the big picture
2. Strings can be dangerous
3. HTML and CGI scripts: select, case, generating HTML
4. A Bigger Project: trip planner
5. structs -- the basics
6. structs -- arrays to model train runs

1. Quick Review

The big picture for C/Unix/CGI programs is:

![Diagram of user, html form, internet, CGI connector, shell script, tools]

2. Strings Can Be Dangerous

Most of the data flowing through this sequence of connections is plain text. C programs store text in strings -- arrays of characters terminated by a null character. Working with strings in C is not as easy as it is in other languages. There are two main types of problems.

String Operations Require Functions

The first problem new C programmers face with strings is that simple assignment, comparison, and concatenation operators do not work with strings. For example:

```c
char s[10], t[10], u[10];

s = "abc";
t = s;
u = "xyz";
if ( s == t )...
t = u + s;
```

do not work. Some do not compile, and some compile but do not do what you want.

Strings are arrays, and loading, comparing, or appending arrays require looping through elements. These functions are performed with the functions:

```c
strcpy(dest, src)
strcmp(str1, str2)
strcat(dest, newstr)
```

Read the man page on these functions for details. We explore the program called string-ops.c to see how these operations work and how the functions work.
String Overruns

The most common C programming error is probably unchecked buffer overflows. These bugs cause programs to crash and allow computer vandals to break into computer systems. Understanding why they occur and how to prevent them is an essential skill. We explore two example programs:

memdemo1.c

```c
#include <stdio.h>

/*
 * memdemo - show how array over-runs are easy
 * variables are allocated on the stack in order of request
 */
main()
{
    char c;
    int i;
    char a[6];
    char b[6];
    while(1)
    {
        printf("c=%c i=%d a=%s b=%s a[-6]=%c b[10]=%c\n",
                c, i, a, b, a[-6], b[10]);
        printf("enter a: ");
        scanf("%s", a);
        printf("c=%c i=%d a=%s b=%s a[-6]=%c b[10]=%c\n",
                c, i, a, b, a[-6], b[10]);
        printf("enter b: ");
        scanf("%s", b);
        printf("c=%c i=%d a=%s b=%s a[-6]=%c b[10]=%c\n",
                c, i, a, b, a[-6], b[10]);
        printf("enter i: ");
        scanf("%d", &i);
    }
}
```

Notice that this function contains four variables, two of them char arrays. What happens to these variables if we enter more characters than the length of the array? Data from one string appears in other variables! What is going on?

Local variables are stored in memory on the stack. When a function begins, space needed for the variables is allocated on the stack by moving the stack pointer down the correct amount. When the function ends, the stack pointer moves back up, effectively deallocating the local variables, although leaving their contents unchanged.

Most C compilers allocate variables in the order requested. Therefore the six chars for a appear above the six chars for b. If we put too many characters in b, they run over into a. If we put enough extra characters into b, they run all the way into c and i and maybe farther.

memdemo2.c:

```c
#include <stdio.h>

/*
 * memdemo2.c -- show how variables in functions can affect
 * variables in calling functions
 */
main()
{
    char a[10] = "abc";
    printf("before function call: a is %s\n", a);
    f();
}
```
printf("after function call: a is %s\n", a);
}

f()
{
    char b[3];
    int i;
    int c;

    printf("in f, please enter string for b: ");
    for(i=0; (c=getchar())!='\n'; i++)
        b[i] = c;
    b[i] = '\0';
    printf("in f, b is now %s\n",b);
}

What about functions and variables? Can overruns in a function affect memory in the calling function? Yes. memdemo2.c shows this.

**Preventing Overruns**

Only you can prevent buffer overruns. Always check indexes as you load arrays. Never assume the size will be 'large enough' and never assume the source of data will know how much space is available.

3. **HTML and CGI scripts: select, checkboxes, case, generating HTML**

We now return to our train schedule website. As we did last time, we list some improvements to make, and then we learn new techniques that allow us to make the improvements.

We now have three CGI programs, one to find times of trains through a station, one to print a schedule, and one to print the names of stations. We continue to improve this site. This week we add the following:

1. Easier Input -- User selects a station from a list
2. Nicer Reports -- use HTML on the response page
3. Better Integration -- include a form on response page

3.1. **Easier Input -- Train-Times with a Select List**

*Problem: User has to type in name of station:* The train-times page works, but the user has to know the names of the stations. The user could of course click on the stationnames link and copy and paste a string from there, but that is not user-friendly.

On the mbta.com website, the train schedule page uses the select-from-a-list input for direction and day. We now modify the form for train-times so the user can select a station from a list rather than have to type the name. To do this, we replace the text input area with a `<select>` input control. A select has the format:

```html
<select name='NAME'>
    <option value='V1'>Str1</option>
    <option value='V2'>Str2</option>
    ...
</select>
```

The `value` attribute of the option items is not required. If omitted, the value of the selection is the string. We replace the text input with a select, and to demonstrate the use of the value attribute, we change the when choice to use a select. The result is:

```html
<!-- train-times3.html uses select for station and day -->
<!-- the connector calls train-times-tabbed for nicer output -->

<body style='background-color: lightblue'>
Find train times for a station
<form action="train-times-tabbed.cgi" method="get">
    <p style='background-color: lightgreen; margin: 12px; padding: 10px; width: 450px'>
        Station: 
```
Notice that this calls train-times-tabbed.cgi which connects the request to train-times-tabbed, which uses rmtags and semi2tab to produce nicer, but still plain text output. Let us focus on a nicer output format.

3.2. Nicer Reports -- use HTML on response

Problem: plain text is boring  Our train-times4.html has a much nicer user input form using selects and CSS styles for layout, but the output is plain text. We can use the same input form with a new back-end. We make a connector called train-times-html.cgi which calls train-times-html which adds HTML tags to the tabbed output. How does that work?

```
#!/bin/sh
# train-times3.cgi
# processing script for train-times3.html
# this one produces html
eval $(./qryparse)
# announce type
echo "Content-Type: text/html"
echo ""
# build up some strings for response
case "$dir" in
  i*) DIR="Inbound" ;;
  o*) DIR="Outbound" ;;
  *) DIR="Inbound" ;;
esac
TITLE="$DIR Trains through $station"
# generate html
echo "<html><head><title>$TITLE</title></head>
<body style='background-color: darkblue; color: white'>
<b>$TITLE</b>
<hr/>
./train-times-tabbed "$station" "$dir" "$when" | ./capitalize | sed 's/$/<br>/'
# print a footer
echo "<hr/>
echo "<span style='font-size: small'>Generated by $1 on "
date
echo "</span>"
```

This CGI program demonstrates three main ideas:

a. The case control structure in shell scripts
b. The technique of using `echo` to output HTML tags
c. The use of `sed` to modify data flowing through a pipe

The `case` control structure works well with radio buttons on a webpage. The user picks one of several items, and the script can take different action based on the choice.

The Unix tool `sed` is a "stream editor". `sed` is designed to be used in pipelines to modify text as it flows through the pipeline. A common use of `sed` is for text replacement. The syntax:

```
s/old/new/  # substitute command
```

is the substitute command. The symbol `\$` stands for end of line.

Producing nice looking output is easy. First tell the browser you are sending content of type `text/html`. Then use `echo` to print out the introductory html for the page. Include a title, headers, style requests. Then generate the data and wrap it in html tags to make it look the way you want.

### Nicer Output for trainsched

Let’s apply this technique of wrapping plain text in HTML tags to make output nicer to the `trainsched` page. For version 2 of trainsched, we send a train number and a y/n answer for capitalization. The form is ok, but the output could be nicer. Again we write a new CGI script, this time called `trainsched3.cgi`:

```bash
#!/bin/sh
#
# trainsched3.cgi
# connector for getting train schedule
# v3: added html format
#  v2: this one uses and if then else to capitalize or not
# eval $(./qryparse)
TITLE="Schedule for Train $trainnumber"
# announce content type
echo "Content-Type: text/html"
echo ""  
# start of html, include title and body start
echo "<html>
<head><title>$TITLE</title></head>
<body style='background-color: darkblue; color: white;'>
<div style='font-weight: bold'>$TITLE</div>
<hr />
"  

# content here
if test "$caps" = "y"
  then
  ./trainsched2 "$trainnumber" | ./capitalize | sed 's/$/<br>/'
else
  ./trainsched2 "$trainnumber" | sed 's/$/<br>/'
fi

# footer and end
./footer $0
echo "</body></html>"
```

Again, we see the same three main techniques. First, tell the browser we are sending back content of type `text/html`.
text/html. Then use echo to send back any introductory html to get the page set up. Finally, generate the data and wrap it in any html tags you need to make it look the way you want.

a. Setting a variable TITLE to be used later in the script
b. Calling a separate script (footer) to produce commonly used code

3.3. Better Integration -- include hyperlinks on response page

This page is certainly looking better. But, the page is cumbersome to use. We pick a station and get a list of trains. To see the schedule for that page, we then need to go to the trainsched.html page to enter the train number. It would be nicer if we could enter the train number from the result page, or even better, just click on the train number.

For our first attempt, we put the form to request a schedule on the result from the train time searcher. This is train-times5.html and train-times-html-form.cgi:

```bash
#!/bin/sh
#train-times-html-form.cgi
#processing script for train-times5.html
#this one produces html and a form for trainsched

eval $(./qryparse)

echo "Content-Type: text/html"
echo ""

case "$dir" in
  i*) DIR="Inbound" ;;
  o*) DIR="Outbound" ;;
  *) DIR="Inbound" ;;
esac
TITLE="$DIR Trains through $station"

# generate html

echo "<html><head><title>$TITLE</title></head>
<body bgcolor='darkblue' text='white'>
<b>$TITLE</b>
<hr/>
./train-times-tabbed "$station" "$dir" "$when" | ./capitalize | sed 's/$/<br>/'
# include the trainsched form

echo "
<form action='trainsched3.cgi'>
  Train number please?
  <input type='text' name='trainnumber' size='10'>
</form>
Capitalize Names: <input type='radio' name='caps' value='y'>Yes
  <input type='radio' name='caps' value='n'>No
</form>"

./footer "$0"
```

This works better than the old method, but it is still not the best solution. Think about how to make the train numbers clickable links that call up the schedule directly.
4. A Bigger Project: A trip Planner

Our site is now looking better, and allows users to search for trains to places they want to visit. A more powerful feature would be the trip planner. The user would select a starting station and an ending station, and the program would figure out which trains to take to get from one place to another.

We shall discuss this problem over several weeks. We start with an html page that allows the user to select two stations.

Here is the form:

```html
<html>
<head><title>Trip Planner Version 1</title></head>
<body>
<center>Trip Planner Version 1</center>
<form action='trip_plan1.cgi'>
Starting station:
<select name='startstn'>
<option>abington
<option>anderson/woburn
... 
<option>wyoming hill
<option>yawkey
</select>
Ending station:
<select name='endstn'>
<option>abington
<option>anderson/woburn
... 
<option>wyoming hill
<option>yawkey
</select>
<input type='submit' value='find trains'>
</form>
</center>
</body>
</html>
```

But what happens when we get these two names? What logic, what tools, what pipeline can we use?

There is not a simple solution to a general trip planner. To figure out how to get from one place to another, we need some way to store and process a train line and figure out when trains get to which stops. There are no standard unix tools to manage this, so we shall have to write some new C code. Our new C code will need to record train trips. A train trip is a sequence of stops. A stop is a station, a time, a train number. Therefore we need a way to create a way to store a sequence of stops.

C has a data type perfect for representing a stop: the struct.

5. Structs - the basics

Say we wanted to record a train trip in a C program. A trip is a sequence of stops. Our datafile records for each stop a a train number, a direction, a time, a station, a day, and a line. Say we wanted to record for each stop the station, the direction, the train number, and the time. C provides the struct as a way to store several values in one container. We can define a new data type called a struct tstop by writing this code:

```c
struct tstop { 
    char station[SLEN]; /* name of station */
    char dir; /* direction 'i' or 'o' */
    char tnum[NLEN]; /* the train number */
    int hr, min; /* stop time */
};
```

This defines what a struct tstop looks like. This is a blueprint, not a house. To instantiate actual variables,
we write:

```c
struct tstop s1, s2; /* create trainstops */
```

And we can work with the members of these structs using:

```c
strcpy(s1.station, "salem");
s1.dir = 'o';
s1.hr = 9;
s1.min = 40;
```

**Operations on Structs**

The three basic operations one may perform on structs are:

- **access** Use the dot operator to select members in a struct. In the examples above, we see how to refer to a string, a char, and ints. The name of the struct is on the left, and the name of the member is on the right.

- **assign** Structs, unlike arrays, may be copied with the = operator. Thus, s2 = s1, will copy the entire struct. No special function is needed.

- **functions** Structs are passed to functions by value and may be returned by value.

See structdemo1.c for details:

```c
#include <stdio.h>
/* structdemo1.c -- show basic struct operations */
#define SLEN 100
#define NLEN 10
/* type declaration -- a blueprint, not a house */
struct tstop {
    char station[SLEN];
    char dir;
    char tnum[NLEN];
    int hr, min;
};
main()
{
    struct tstop s1, s2; /* two houses */
    strcpy(s1.station, "salem");
    s1.dir = 'o';
    s1.hr = 9;
    s1.min = 45;
    strcpy(s1.tnum, "114");
    show_stop("struct s1:\n", s1);
    s2 = s1;
    show_stop("struct s2:\n", s2);
}
/*
 * print stop info
 */
show_stop(char msg[], struct tstop s)
{
    printf("%s", msg);
    printf("TR=%s;dir=%c;TI=%02d:%02d;stn=%s\n",
            s.tnum, s.dir, s.hr, s.min, s.station);
}
```

6. structs -- arrays to model train runs

A train trip is a sequence of stops. We model that with an array of stops:

```c
struct tstop trip[NUM_STOPS];
```

Each element in this array is of the form trip[i] and is a struct tstop. The name of the station, for example,
is trip[i].station. We now write a program to read data from the user, build an array, then print the array.

```c
#include <stdio.h>

/* readtrip.c
 * shows: how to use an array of structs
 * use: reads in one train trip worth of stops
 */

#define MAXSTOPS 100
#define LINELEN 200
#define SLEN 40
#define NLEN 10
#define YES 1
#define NO 0

struct tstop {
    char station[SLEN];
    char dir;
    char tnum[NLEN];
    int hr, min;
};

main()
{
    struct tstop trip[MAXSTOPS];
    int n = 0;
    while( n<MAXSTOPS && read_in_stop(trip, n) == YES )
        n++;
    show_stops(trip, n);
}

/*
 * prompt user for items, load the struct at pos
 * Return NO for no more data, YES for more
 */
int read_in_stop(struct tstop a[], int pos)
{
    char line[LINELEN];
    int hh, mm;
    if ( getline("station:", line, LINELEN) == NO )
        return NO;
    strcpy(a[pos].station, line);
    if ( getline("direction: ", line, LINELEN) == NO )
        return NO;
    a[pos].dir = line[0];
    if ( getline("number: ", line, LINELEN) == NO )
        return NO;
    strcpy(a[pos].tnum, line);
    if ( getline("time (hh:mm): ", line, LINELEN) == NO )
        return NO;
    if ( 2 == sscanf(line, "%d:%d", &hh, &mm) ){
        a[pos].hr = hh;
        a[pos].min = mm;
    }
    return YES;
}

int getline(char prompt[], char buf[], int len)
{
    printf("%s", prompt);
    if ( fgets(buf, len, stdin) == NULL )
        return NO;
    buf[strlen(buf)-1] = ' ';
    return YES;
}
```
/ 
* list all the stops recorded in the array of structs 
*/
show_stops(struct tstop trip[], int len)
{
    int i;
    struct tstop s;
    for(i=0; i<len; i++)
    {
        printf("Stop %d: ", i);
        s = trip[i];
        printf("TR=%s;dir=%c;TI=%02d:%02d;stn=%s\n", 
                s.tnum, s.dir, s.hr, s.min, s.station);
    }
}