Computer Science 50 Week 9 Monday: October 31, 2011
Fall 2011 Andrew Sellergren
Scribe Notes

Contents

1 Announcements and Demos (0:00–4:00)

- David is on Team Bowden! And he has Bowden Fever!
- CS50 Dinner will be held this Wednesday at 6 p.m. RSVP if you'd like to come!
- CS50 Finance is the first of two web-based problem sets which you will complete over the next two weeks. We've provided you with a framework which you might find useful for the Final Project, as well.
- Register for seminars if you haven't already. This year, we are featuring more seminars than ever before, including a few on Windows mobile programming presented by Microsoft themselves. RIMM has also kindly donated some BlackBerry Bolds for those interested in BlackBerry Final Projects.

2 PHP (4:00–54:00)

2.1 From C to PHP

- PHP is fundamentally different from C in that it is *interpreted*, not *compiled*. That means it isn't converted to binary until it is actually run. One advantage of this is not having to recompile your program every time you make a change. Another advantage of interpreted languages is that they tend to be higher level meaning you don't have to worry about pointers and memory management because they are built in.
- The PHP manual will prove to be a valuable resource to you as you begin this next problem set.
- Syntactically, PHP is very similar to C. Conditions, Boolean expressions, loops, and switches all look the same in PHP as they do in C. In fact, in PHP, you can also use strings as your switch variable. Arrays in PHP have no finite size so you can add to them as needed. Variables in PHP are prefixed with $. PHP is loosely typed, so you don't have to specify the type of your variables. In fact, the type of your variables can change on the fly.
- One additional feature of loops in PHP is the `foreach` construct:

  ```
  foreach ($array as $element)
  {
      // do this with $element
  }
  ```

  This syntax allows you to loop over all of the elements in an array without needing to know how many there are.

- PHP offers associative arrays as built in. For Problem Set 6, you might have implemented a hash table in order to store the dictionary. An associative array is essentially a hash table: its keys can be of any type, not just integers as in C.

2.2 Re-implementing Problem Set 6

- Turns out that Problem Set 6 only takes a matter of minutes to implement in PHP. Let's give it a shot:

```
<?

$size = 0;

$dictionary = array();

function load($d)
{
    global $size;
    global $dictionary;

    if (!file_exists($d))
        return false;
    if (!is_readable($d))
        return false;

    $words = file($d);
    foreach($words as $word)
    {
        $word = chop($word);
        $dictionary[$word] = true;
        $size++;
    }
}

?>
```

- In order to use global variables within a function, we must redeclare them with the keyword `global`. The `file_exists` and `is_readable` functions do some error checking for us and `words` reads our file into memory, creating an array in which each line is an element. Then we use the `foreach` construct to loop through our array and insert each word into our associative array (after using `chop` to trim the newline character from the end), incrementing `$size` as we do. That's it for `load`!

- Note that the opening and closing tags can also be written as `<?php` and `?>`. Web servers must be configured to recognize the short tags we use above, but most are.

- The `size` function is even easier to implement:

  ```
  function size()
  {
      global $size;
      return $size;
  }
  ```

- Finally, `check` is perhaps the most interesting function to implement, although it too is quite simple:

  ```
  function check($w)
  {
      global $dictionary;
      $w = strtolower($w);
      return $dictionary[$w];
  }
  ```

  All we need to do is convert the word to all lowercase and check whether it is present as a key in our associative array. Done!

- Why did we bother implementing Problem Set 6 in C, then? Although PHP is much faster than C for development, it is much slower than C for execution. In general, interpreted languages are much slower than compiled languages because they must be translated into binary during every execution whereas compiled languages are already translated into binary after the compilation step. When we really care about performance, as we did in Problem Set 6, we're willing to spend more time writing code.

- To prove to you that PHP is slower than C, we'll run these implementations side by side. Whereas the PHP implementation takes about 0.5 seconds to spellcheck `austinpowers.txt`, the C implementation takes only 0.01 seconds. Note that at the top of `speller`, our PHP version of `speller.c`, there is a *shebang*, a sharp (`#`) followed by a bang (`!`) followed by a path to a program (`/usr/bin/php/`) which enables this code to be executable from the command line.

- Why write in PHP at all then? In the context of web programming, the execution time of a program on the server is not necessarily the bottleneck: the web request's trip to and from the client is the more likely culprit. Moreover, most web servers offer the ability to cache PHP meaning that subsequent executions of the same script will be much faster than the first.

2.3 Frosh IMs

2.3.1 `register0.php`

- Recall from last time that the `action` attribute of a `form` element specifies where the form is submitted. In `froshims0.php`, we specify `register0.php` for the `action` attribute. The `method` attribute can be either `get` or `post` depending on if we want the parameters passed in the URL or more securely in the HTTP headers.

- `register0.php` looks as follows:

  ```
  <?
      /**************************************************************
       * register0.php
       *
       * Computer Science 50
       * David J. Malan
       *
       * Dumps contents of $_POST.
       **************************************************************/

  ?>

  <!DOCTYPE html>

  <html>
    <head>
      <title>Frosh IMs</title>
    </head>
    <body>
      <pre>
        <? print_r($_POST); ?>
      </pre>
    </body>
  </html>
  ```

  `print_r` is a useful diagnostic function that recursively prints the contents of a variable, in this case, the special superglobal variable `$_POST`, which contains the parameters passed from our form via the POST method. If we actually submit the form, we might get something like this:

```
Array
{
    [name] => David
    [captain] => on
    [gender] => M
    [dorm] => Matthews
}
```

`on` is the value that's sent for a checked checkbox.

- In addition to `$_POST`, there are a number of superglobal variables in PHP:
  - `$_COOKIE`
  - `$_GET`
  - `$_POST`
  - `$_SERVER`
  - `$_SESSION`

  `$_GET` is the equivalent of `$_POST` for GET requests. `$_SERVER` stores the values of some server variables as well as the user's IP address and browser information, and `$_COOKIE` stores the user's cookie, which we'll talk about shortly. `$_SESSION` is a variable that allows a website to remember some information about a user until she navigates away. By default, HTTP is a stateless protocol, meaning that as soon as a webpage is downloaded from a server, the connection to the client is severed and the server more or less forgets about the state of the client. To implement something like a shopping cart, though, we need some way to keep track of the client's state in between connections to the server. This is where the `$_SESSION` variable comes in: we can store the contents of the shopping cart in this server-side variable so that if the same user reconnects to the server, we'll be able to quickly recall what she was planning to buy.

2.3.2 register1.php

- Recall from last time `register1.php` which implemented some error checking on the user's form submission:

  ```
  <?
      /****************************************************************
       * register1.php
       *
       * Computer Science 50
       * David J. Malan
       *
       * Implements a registration form for Frosh IMs.  Redirects
       * user to froshims1.php upon error.
       ***************************************************************/

      // validate submission
      if (empty($_POST["name"]) || empty($_POST["gender"]) || empty($_POST["dorm"]))
      {
          header("Location: http://localhost/~jharvard/froshims/froshims1.php");
          exit;
      }

  ?>

  <!DOCTYPE html>

  <html>
    <head>
      <title>Frosh IMs</title>
    </head>
    <body>
      You are registered!  (Well, not really.)
    </body>
  </html>
  ```

  The `empty` function returns true if there's nothing in the variable it is passed. The logic above, then, checks if the user failed to pass a name, gender, or dorm via the form. If he did, then he will be redirected back to the form using the Location header. Notice that the HTML at the bottom of the file is only sent to the user if he did pass name, gender, and form: if he didn't, he would be redirected before it was printed out.

2.3.3 register2.php

- `register2.php` demonstrates how we can commingle PHP and HTML:

```
<?
    /***********************************************************************
     * register2.php
     *
     * Computer Science 50
     * David J. Malan
     *
     * Implements a registration form for Frosh IMs.  Informs user of
     * any errors.
     **********************************************************************/
?>

<!DOCTYPE html>

<html>
  <head>
    <title>Frosh IMs</title>
  </head>
  <body>
    <? if (empty($_POST["name"]) || empty($_POST["gender"]) ||
           empty($_POST["dorm"])): ?>
      You must provide your name, gender, and dorm!
      Go <a href="froshims2.php">back</a>.
    <? else: ?>
      You are registered!  (Well, not really.)
    <? endif ?>
  </body>
</html>
```

The colon at the end of our if condition tells PHP to execute whatever's below (and only what's below) even though we're exiting PHP mode with a closing `?>` tag.

2.3.4 `register3.php`

- We gave you a preview last time of `register3.php`, which sends an e-mail to David upon form submission:

```
<?
    /***********************************************************************
     * register3.php
     *
     * Computer Science 50
     * David J. Malan
     *
     * Implements a registration form for Frosh IMs.  Reports registration
     * via email.  Redirects user to froshims3.php upon error.
     **********************************************************************/

    // validate submission
    if (!empty($_POST["name"]) && !empty($_POST["gender"]) && !empty($_POST["dorm"]))
    {

        $to = "malan@cs50.net";
        $subject = "Registration";
        $body = "This person just registered:\n\n" .
         $_POST["name"] . "\n" .
         $_POST["captain"] . "\n" .
         $_POST["gender"] . "\n" .
         $_POST["dorm"];
        $headers = "From: malan@cs50.net\r\n";
        mail($to, $subject, $body, $headers);
    }
    else
    {
        header("Location: http://localhost/~jharvard/froshims/froshims3.php");
        exit;
    }
?>

<!DOCTYPE html>

<html>
  <head>
    <title>Frosh IMs</title>
  </head>
  <body>
    You are registered!  (Really.)
  </body>
</html>
```

The dot operator concatenates strings. The `mail` function actually sends the e-mail.

2.3.5 froshims4.php

- Thus far, our forms have submitted to a separate page. `froshims4.php`, however, submits to itself:

```
<?
    /*********************************************************************
     * froshims4.php
     *
     * Computer Science 50
     * David J. Malan
     *
     * Implements a registration form for Frosh IMs.  Submits to itself.
     ********************************************************************/

    // if form was actually submitted, check for error
    if (isset($_POST["action"]))
    {
        if (empty($_POST["name"]) || empty($_POST["gender"]) || empty($_POST["dorm"]))
            $error = true;
    }
?>

<!DOCTYPE html>

<html>
  <head>
    <title>Frosh IMs</title>
  </head>
  <body>
    <div style="text-align: center">
      <h1>Register for Frosh IMs</h1>
      <? if ($error): ?>
        <div style="color: red">You must fill out the form!</div>
      <? endif ?>
      <br><br>
      <form action="froshims4.php" method="post">
        <table style="border: 0; margin-left: auto;
          margin-right: auto; text-align: left">
          <tr>
            <td>Name:</td>
            <td><input name="name" type="text"></td>
          </tr>
          <tr>
            <td>Captain:</td>
            <td><input name="captain" type="checkbox"></td>
          </tr>
          <tr>
            <td>Gender:</td>
            <td>
              <input name="gender" type="radio" value="F"> F
              <input name="gender" type="radio" value="M"> M
            </td>
          </tr>
          <tr>
            <td>Dorm:</td>
            <td>
              <select name="dorm">
                <option value=""></option>
                <option value="Apley Court">Apley Court</option>
                <option value="Canaday">Canaday</option>
                <option value="Grays">Grays</option>
                <option value="Greenough">Greenough</option>
                <option value="Hollis">Hollis</option>
                <option value="Holworthy">Holworthy</option>
                <option value="Hurlbut">Hurlbut</option>
                <option value="Lionel">Lionel</option>
                <option value="Matthews">Matthews</option>
                <option value="Mower">Mower</option>
                <option value="Pennypacker">Pennypacker</option>
                <option value="Stoughton">Stoughton</option>
                <option value="Straus">Straus</option>
                <option value="Thayer">Thayer</option>
                <option value="Weld">Weld</option>
                <option value="Wigglesworth">Wigglesworth</option>
              </select>
            </td>
          </tr>
        </table>
        <br><br>
        <input name="action" type="submit" value="Register!">
      </form>
    </div>
  </body>
</html>
```

By submitting to a separate page, we either forced the user to click a link to go back or bounced him back automatically without any explanation. You probably already know how frustrating an experience this can be when you go back and all of your previously entered form data is gone.

- To make for a better user experience, we can do error checking before leaving the form page. In this example, our submit button has a `name` attribute set to `action`. At the top of the page, before any HTML is outputted, we check if the `action` key in the `$_POST` array is set. If it's set, it's because the user has already submitted the form, so we go ahead and validate his name, dorm, and gender. If any of these variables is empty, we set a variable named `$error` to `true` which we'll use later in the HTML in order to output an error message if necessary. Although the form fields are still cleared, this is arguably a better user experience than having to navigate back.

- As a sidenote, PHP has errors, warnings, and notices. In C, we made all three of them cause compilation to fail. In PHP, we will disable notices.

2.3.6 `froshims5.php`

- `froshims5.php` also submits to itself, but manages to keep the name field populated upon error:

```
<?
    /***********************************************************************
     * froshims5.php
     *
     * Computer Science 50
     * David J. Malan
     *
     * Implements a registration form for Frosh IMs.  Submits to itself.
     * Pre-populates name field upon error.
     **********************************************************************/

    // if form was actually submitted, check for error
    if (isset($_POST["action"]))
    {
        if (empty($_POST["name"]) || empty($_POST["gender"]) || empty($_POST["dorm"]))
            $error = true;
    }
?>

<!DOCTYPE html>

<html>
  <head>
    <title>Frosh IMs</title>
  </head>
  <body>
    <div style="text-align: center">
      <h1>Register for Frosh IMs</h1>
```

```
<? if ($error): ?>
  <div style="color: red">You must fill out the form!</div>
<? endif ?>
<br><br>
<form action="froshims5.php" method="post">
  <table style="border: 0; margin-left: auto;
     margin-right: auto; text-align: left">
    <tr>
      <td>Name:</td>
      <td><input name="name" type="text"
        value="<?= htmlspecialchars($_POST["name"]) ?>"></td>
    </tr>
    <tr>
      <td>Captain:</td>
      <td><input name="captain" type="checkbox"></td>
    </tr>
    <tr>
      <td>Gender:</td>
      <td>
        <input name="gender" type="radio" value="F"> F
        <input name="gender" type="radio" value="M"> M
      </td>
    </tr>
    <tr>
      <td>Dorm:</td>
      <td>
        <select name="dorm">
          <option value=""></option>
          <option value="Apley Court">Apley Court</option>
          <option value="Canaday">Canaday</option>
          <option value="Grays">Grays</option>
          <option value="Greenough">Greenough</option>
          <option value="Hollis">Hollis</option>
          <option value="Holworthy">Holworthy</option>
          <option value="Hurlbut">Hurlbut</option>
          <option value="Lionel">Lionel</option>
          <option value="Matthews">Matthews</option>
          <option value="Mower">Mower</option>
          <option value="Pennypacker">Pennypacker</option>
          <option value="Stoughton">Stoughton</option>
          <option value="Straus">Straus</option>
          <option value="Thayer">Thayer</option>
          <option value="Weld">Weld</option>
          <option value="Wigglesworth">Wigglesworth</option>
        </select>
      </td>
```

```
          </tr>
        </table>
        <br><br>
        <input name="action" type="submit" value="Register!">
      </form>
    </div>
  </body>
</html>
```

The `htmlspecialchars` function ensures that the user's input is displayed literally and doesn't muck with our website's tags. In this case, we're printing what the user inputted for his name, if anything, as the `value` attribute of the name field.

- The `<?=` syntax is shorthand for `<? echo` or `<? print`.
- If we don't call the `htmlspecialchars` function before printing the user's input, his input can actually mess up the source code of our webpage. For example, a user might input `''><b>HAHAHA` and add some laughter to our page outside the bounds of the form. Worse, a malicious user might inject a `script` tag containing some JavaScript that would steal your cookie for a particular website. If we do call the `htmlspecialchars` function, then this same input will be converted to `&quot;&gt;&lt;b&gt;HAHAHA` using so-called HTML entities in place of angle brackets and quotation marks.

3 Database-driven Websites (54:00–73:00)

- Before we examine the most compelling version of our freshman intramurals registration page, let's talk about the database behind it. A database is a storage platform that organizes persistent data into tables. You can think of a database and its tables as an Excel file and the spreadsheets within it, respectively.
- Database engines abound. There are Microsoft Access, MySQL, Oracle, and Microsoft SQL Server just to name a few. We'll be using MySQL because it's free and open-source (i.e. the source code is publicly available and modifiable). Facebook is one of MySQL's largest proponents and has shown that MySQL can be scalable enough to run enterprise applications. MySQL is generally very popular and is widely supported by domain hosts. To interact with our database server, we'll be using phpMyAdmin, another excellent piece of free, open-source software.

3.1 Problem Set 7

- For Problem Set 7, you will begin with a `users` table that contains columns for user ID, username, and password hash. Why a password hash instead of the password itself? By storing the hash instead of the password itself, we limit the damage that a malicious user can do by gaining access to this table; even if he is able to download all the rows, he still hasn't stolen any passwords since they are one-way encrypted.

- SQL, or structured query language, isn't exactly a programming language. Rather, it is a language that enables us to get at the data that is stored in databases. For our purposes, SQL boils down to four basic commands:
  - `INSERT`
  - `SELECT`
  - `UPDATE`
  - `DELETE`
- To open phpMyAdmin, navigate to `http://localhost/phpmyadmin` on the Appliance. You'll be prompted for a username and password which, by default, are `jharvard` and `crimson`, respectively.
- The specification for Problem Set 7 will instruct you to execute some SQL commands within phpMyAdmin which will create the aforementioned `users` table.

3.2 More with Frosh IMs

- To organize our frosh IMs data, we'll begin by creating a new database (by navigating to the home page and clicking the Databases tab) which we'll call `jharvard_week9`. Note that the user `jharvard` has privileges to create any database beginning with `jharvard_`. When we click the Create button, we'll see that the following SQL command was executed:

  ```
  CREATE DATABASE jharvard_week9;
  ```

  Herein lies the advantage of phpMyAdmin: it allows you to point and click in order to execute somewhat arcane SQL commands.
- Intuitively, we can organize our frosh IMs registration information into a single table called `registrants` that contains columns for name, gender, dorm, and captainship. For performance reasons, MySQL requires strict data typing, so we need to decide ahead of time what data types these four columns will have. Let's go with `VARCHAR` (a type of string which can have variable length) for name, with a maximum length of 128, `BOOLEAN` for captain, `CHAR` for gender (although there are many different data types that could work here) with a maximum length of 1, and `VARCHAR` for dorm, with a maximum length of 128.
- The string type `CHAR` has the downside that if you specify a length of 64, every entry in this column will take up 64 bytes even if it's not 64 characters long. `VARCHAR` optimizes by only storing the exact number of

bytes that are needed. The tradeoff, however, is that VARCHAR is slower for lookups. If MySQL knows exactly where each string begins and ends, as with a CHAR type, it can jump quickly between entries as it searches.

- Most of the other dropdown menus displayed during the creation of a table aren't all that important. In the Attributes menu, we could choose UNSIGNED for our an integer column if we don't want to waste a bit to store negative numbres. The Index menu allows us to specify a column for which lookups will be optimized. If you designate a column as an INDEX, MySQL will build a tree structure under the hood so that finding a particular value in that column is more efficient. The Null checkbox indicates where the special value NULL is allowed in this column.

- When we click the Save button, the table will be created, but we'll also see the actual SQL statement which was executed. By the way, don't fret if you see Collation defined as something like latin_swedish_ci. This is just because the creators of MySQL were Swedish.[1] Thankfully, the Swedish character set overlaps with the English character set.

3.2.1 register8.php

- Let's take a look at how we will interact with our newly created database using PHP in register8.php:

```
<?
    /***********************************************************************
     * register8.php
     *
     * Computer Science 50
     * David J. Malan
     *
     * Implements a registration form for Frosh IMs.  Records registration
     * in database.  Redirects user to froshims8.php upon error.
     **********************************************************************/

    // validate submission
    if (empty($_POST["name"]) || empty($_POST["gender"]) || empty($_POST["dorm"]))
    {
        header("Location: http://localhost/~jharvard/froshims/froshims8.php");
        exit;
    }

    // connect to database
    mysql_connect("localhost", "jharvard", "crimson");
    mysql_select_db("jharvard_week9");
```

[1] Bork bork bork!

```
    // scrub inputs
    $name = mysql_real_escape_string($_POST["name"]);
    if ($_POST["captain"])
        $captain = 1;
    else
        $captain = 0;
    $gender = mysql_real_escape_string($_POST["gender"]);
    $dorm = mysql_real_escape_string($_POST["dorm"]);

    // prepare query
    $sql = "INSERT INTO registrants (name, captain, gender, dorm)
     VALUES('$name', $captain, '$gender', '$dorm')";

    // execute query
    mysql_query($sql);
?>

<!DOCTYPE html>

<html>
  <head>
    <title>Frosh IMs</title>
  </head>
  <body>
    You are registered!  (Really.)
  </body>
</html>
```

First, we connect to the database server by providing our login credentials to the `mysql_connect` function. Second, because there will be multiple databases on this server, we must select ours by calling the `mysql_select_db` function. Third, we need to scrub the data that the user has provided. That is, because there are a lot of ways that malicious users can sabotage our database if we blindly pass their inputs to the database server, we need to go through a process of cleaning those inputs. We do this with the `mysql_real_escape_string` function. Fourth, we will create a SQL query and execute it using the `mysql_query` function. Assuming all of these steps are successful, the user's data will be inserted into our MySQL database.

- The format of a MySQL `INSERT` statement is roughly as follows: `INSERT INTO` followed by the table name, a comma-separated list of columns in parentheses, the keyword `VALUES`, and a comma-separated list of values in parentheses.

- Notice that to print the value of a variable like `$gender`, we need only write that variable's name between quotation marks. It will be *interpolated*, or automatically replaced with its value.

- If we navigate to `froshims8.php`, which is nearly identical to earlier versions of the form, and enter in our data, we see a message indicating that we've successfully registered. The real test comes, however, when we check the contents of our database using phpMyAdmin. And ta-da, it's there!