Week 8

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. Announcements

- We're excited to welcome back CS50's own Ramon Galvan, who's been working in Hollywood on a TV show called Colony!
- The end is near! Meaning that there's really not all that much left of CS50:
  # We have just three problem sets remaining - you have Problem Set 6 right now, and after that you'll just have two more.
    # Problem Set 6 bridges our command-line world thus far to the world of web programming by having you implement a web server. We give you significant amounts of distribution code, which is great practice for the real world, where you rarely start a project from scratch - you're usually bolting your code onto something that somebody's already written.
    # Problem Set 7 tasks you with building a dynamic web-based application, a bit like E*Trade.
    # Problem Set 8 will focus on yet another language, known as JavaScript.
  # Final projects are also imminent.
    # Pre-proposal due Monday 11/2 at noon! This is mostly just a sanity check to make sure you're planning something feasible (and so your TF can help you figure out how to adjust your plans if you're not).

    - Proposal and Status Report will come up in a little while; more details as they approach.
    - There will be a CS50 Hackathon for Harvard and Yale students alike.
    - After your implementation is due, there will be two separate CS50 Fairs - one at Harvard and one at Yale.

- **CS50 Seminars** are starting. They can be great stepping stones to a final project, or just to get you familiar with some material beyond the reach of the course itself. If you can't make it to a seminar in person, they will all be streaming live at live.cs50.net[1] and available on-demand afterwards at manual.cs50.net/seminars[2].
- The CS50 Hackathon is coming up, as is the CS50 Fair!
- If you'd like to think about joining CS50's own teaching staff, we'll talk more about this in a bit, but do know this is an option (we'll let you know how to sign up)!

2. PHP

- We can compare C, a low-level language, to PHP (or other higher-level languages), as being like the difference between digging a hole with a spoon versus a big shovel - the shovel, like PHP, lets you take much bigger bites out of the problem, because it comes with much more functionality built in.
- Another possible metaphor: you could use a wrench to hammer in a nail, but the right tool to use is a hammer… C is not necessarily the right tool for most applications (we teach it because it's a great way to explain computer science and programming fundamentals, and its syntax and concepts are extremely transferable to other languages), as there are many languages that are much more specialized to different purposes.
- Recall last week we just wrote a dinky little `hello, world` program, saved it in a file called `hello.php`, and ran it using `php hello.php`.
    - We pass the file to the PHP **interpreter** with this command, and the interpreter understands the code - one line at a time - and executes it.

[1] http://live.cs50.net
[2] http://manual.cs50.net/seminars

  - # Not having to compile your code can speed up the development process significantly, although it comes at the cost of your program being a little bit (and sometimes a lot) slower.
  - # If the computer can read your code directly, it's faster than an interpreter running and telling the computer what to do one step at a time.

- If we open up a file, save it as `hello.php`, and just type the line:

```
print("hello, world\n");
```

- Then we can run this file with `php hello.php` as above:

```
jharvard@ide50:~/workspace $ php hello.php
print("hello, world\n");jharvard@ide50:~/workspace $
```

  - # Hmm - it doesn't seem like our file was actually interpreted! What's wrong?
  - # Well, we need to tell the interpreter that what we gave it was actually PHP code:

```
<?php

    print("hello, world\n");

?>
```

  - # Then we get the desired output:

```
jharvard@ide50:~/workspace $ php hello.php
hello, world
jharvard@ide50:~/workspace $
```

- There's a minor optimization we can make to avoid having to type `php hello.php` every time, and instead be able to run our program as `./hello.php` like we could with our compiled C programs. We prepend our file with the line `#!/usr/bin/env php` as follows:

```
#!/usr/bin/env php
<?php

    print("hello, world\n");

?>
```

# This is called a shebang line (because it starts with a `#`, or sharp, and a `!`, or bang), and it tells the computer what program to use to run the rest of the file by giving it a path to a program on the Linux system.

# `/usr/bin/env` is the path to a program that keeps track of where other programs are on the system, so this shebang line tells the computer to find the PHP interpreter for us and use it to interpret this file.

# Now we can do this:

```
jharvard@ide50:~/workspace $ ./hello.php
bash: ./hello.php: Permission denied
jharvard@ide50:~/workspace $
```

# Hmm. Turns out this is something you're going to see a lot of on the upcoming problem sets; your file isn't executable by default (unlike the compiled programs we had before, where the compiler takes care of the permissions), so we need to set the permissions using `chmod` (which stands for "change mode"):

```
jharvard@ide50:~/workspace $ chmod a+x hello.php
jharvard@ide50:~/workspace $ ./hello.php
hello, world
jharvard@ide50:~/workspace $
```

# `chmod a+x hello.php` means "give all users (`a`) permission to execute (`x`) the file `hello.php`".

# Now it just works, without having to specify the PHP interpreter, because of that shebang line we included.

# We can even change the name of `hello.php` to just `hello` and do exactly what we did before in C:

```
jharvard@ide50:~/workspace $ ./hello
```

```
hello, world
jharvard@ide50:~/workspace $
```

# There's no functional difference, but it's helpful that this way, the user doesn't have to know or care what language your program is written in - they can just run it regardless.

- Let's look at `quote.php`[3]:

```
<?php

    require("functions.php");

    // ensure proper usage
    if ($argc != 2)
    {
        print("Usage: php quote.php symbol\n");
        exit(1);
    }

    // look up stock
    $stock = lookup($argv[1]);

    // report price
    print("1 share of {$stock["name"]} costs {$stock["price"]}.\n");

?>
```

# We start with the now-familiar `<?php`, followed by `require("functions.php");`, which is essentially PHP's version of C's `#include` - it just means "go get the contents of the file `functions.php` and put them here". `functions.php` is just another file we wrote that factors out some functionality.

# In lines 6-10, we check (as we did in many of our C programs) to make sure the user is cooperating and providing input correctly.

# There's no main function, so we don't `return` an error code - instead we `exit(1);`.

[3] http://cdn.cs50.net/2015/fall/lectures/8/m/src8m/quote/quote.php.src

    - `$argc` is just a global variable that PHP provides us with (which means the same thing as `argc` in C).

  - In line 13, we call a function called `lookup`, passing it the second command-line argument.

  - Note in line 16, when we report the stock price, we don't need to use `printf` syntax - we can just include variable values directly in our string by placing the variable name in curly braces. This is called **variable interpolation**. Variable interpolation only works within double-quotes - if you try to do it within single-quotes, you'll see the actual curly braces in your printed string.

  - Let's run this:

```
jharvard@ide50:~/workspace $ php quote.php goog
1 share of Alphabet Inc. costs 717.9178.
jharvard@ide50:~/workspace $ php quote.php msft
1 share of Microsoft Corporation costs 53.81.
jharvard@ide50:~/workspace $ php quote.php yhoo
1 share of Yahoo! Inc. costs 33.355.
jharvard@ide50:~/workspace $ php quote.php fb
1 share of Facebook, Inc. costs 103.18.
jharvard@ide50:~/workspace $
```

- So where is our program actually getting the stock information about these companies? It looks like the magic is happening in that `lookup` function, which it turns out doesn't come from PHP itself, but is implemented in `functions.php`[4]:

[4] http://cdn.cs50.net/2015/fall/lectures/8/m/src8m/quote/functions.php.src

```
/**
 * Returns a stock by symbol (case-insensitively) else false if not found.
 */
function lookup($symbol)
{
    // reject symbols that start with ^
    if (preg_match("/^\^/", $symbol))
    {
        return false;
    }

    // reject symbols that contain commas
    if (preg_match("/,/", $symbol))
    {
        return false;
    }
...
```

  - You needn't worry about the details here, but notice that we literally declare `function lookup($symbol)`, with no type declarations, and then implement the function (starting with some error checking).
  - We'll see this function again in a week on Problem Set 7.

- And note also that we can do the same thing with this program as we did with `hello.php`, allowing it to be run as just `./quote` by including our shebang line from above:

```
jharvard@ide50:~/workspace $ ./quote goog
1 share of Alphabet Inc. costs 717.9178.
jharvard@ide50:~/workspace $
```

- In a web context, we don't need to include the shebang line, because the web server knows to pass files with the `.php` filename extension to a PHP interpreter.

3. MVC

- As we transition to using PHP to write actual web programs that can be served up by a web server, we're going to go into a new framework, or programming paradigm, called **MVC**.

  - There are certain patterns we can follow when writing software, design best practices that have been developed. MVC is one example of how we can best lay out a PHP-based website.

- MVC stands for **Model, View, Controller**, and our code will be split up into model code, view code, and controller code.

- We can oversimplify as follows:

  - **Controller** code is the brains of your site - the loops, conditions, etc, the actual logic.

  - **View** code is the aesthetics of your site - what the user sees.

  - **Model** code is data- and database-related, and we'll talk more about this part later.

- Let's look at an actual example in this week's source code, starting with version 0[5]. In `index.php`, we have:

```
<!DOCTYPE html>

<html>
    <head>
        <title>CS50</title>
    </head>
    <body>
        <h1>CS50</h1>
        <ul>
            <li><a href="lectures.php">Lectures</a></li>
            <li><a href="http://cdn.cs50.net/2015/fall/lectures/0/w/
syllabus/cs50/cs50.html">Syllabus</a></li>
        </ul>
    </body>
</html>
```

  - We can see that the first list item links us to another dynamically generated page, `lectures.php`, which looks like this:

[5] http://cdn.cs50.net/2015/fall/lectures/8/m/src8m/mvc/0/index.php.src

```
<!DOCTYPE html>

<html>
    <head>
        <title>Lectures</title>
    </head>
    <body>
        <h1>Lectures</h1>
        <ul>
            <li><a href="week0.php">Week 0</a></li>
            <li><a href="week1.php">Week 1</a></li>
        </ul>
    </body>
</html>
```

# So when we click the link for Week 0, our browser requests the file `week0.php`, meaning that `week0.php` must contain the controller logic to display the HTML we see when we go to this page.

# Let's look inside `week0.php`[6]:

```
<!DOCTYPE html>

<html>
    <head>
        <title>Week 0</title>
    </head>
    <body>
        <h1>Week 0</h1>
        <ul>
            <li><a href="http://cdn.cs50.net/2015/fall/lectures/0/w/
week0w.pdf">Wednesday</a></li>
            <li><a href="http://cdn.cs50.net/2015/fall/lectures/0/f/
week0f.pdf">Friday</a></li>
        </ul>
    </body>
</html>
```

[6] http://cdn.cs50.net/2015/fall/lectures/8/m/src8m/mvc/0/week0.php.src

- Note that since there are no `<?php` or `?>` tags in any of these three files, none of this code is actually getting interpreted by the PHP interpreter (instead, it just gets spit out as-is, like we saw before).

- But since each week has pretty much identical HTML that we need, we can factor out some of this identical code and have PHP dynamically generate it (better design!)
  - The only differences between `week0.php` and `week1.php` are the titles, headers, and URLs, and they're only different in very specific ways.
  - And even more generally, all our pages (including `index.php` and `lectures.php`) have a very similar structure, with a header that looks almost identical.
  - So in version 1, we can turn `lectures.php`[7] into this:

```
<?php require("header.php"); ?>

<ul>
    <li><a href="week0.php">Week 0</a></li>
    <li><a href="week1.php">Week 1</a></li>
</ul>

<?php require("footer.php"); ?>
```

    - We've factored out the common code at the top and bottom into separate files called `header.php`[8] and `footer.php`[9].
    - `require` just pastes the content of those files in!
    - This doesn't address the remaining redundancy between weeks yet, but it lets us reuse some of our work at the beginning and end (although now all our pages have the same title!)

- Let's take another step forward and fix this all-the-pages-have-the-same-title problem, in version 2 of `index.php`[10]:

[7] http://cdn.cs50.net/2015/fall/lectures/8/m/src8m/mvc/1/lectures.php.src
[8] http://cdn.cs50.net/2015/fall/lectures/8/m/src8m/mvc/1/header.php.src
[9] http://cdn.cs50.net/2015/fall/lectures/8/m/src8m/mvc/1/footer.php.src
[10] http://cdn.cs50.net/2015/fall/lectures/8/m/src8m/mvc/2/index.php.src

```
<?php require("helpers.php"); ?>

<?php renderHeader(["title" => "CS50"]); ?>

<ul>
    <li><a href="lectures.php">Lectures</a></li>
    <li><a href="http://cdn.cs50.net/2015/fall/lectures/0/w/syllabus/cs50/
cs50.html">Syllabus</a></li>
</ul>

<?php renderFooter(); ?>
```

# Now instead of using `require` to insert the header and footer, we're `require`ing just a single file, called `helpers.php` (which just contains a bunch of helper functions that we've written).

# In line 3 and line 10, we're calling PHP functions `renderHeader()` and `renderFooter()`, both of which we wrote in `helpers.php`.

# The argument to `renderHeader()` is an **associative array**, if you'll recall from last week - we're passing in a single key-value pair, where the key is "title" and the value is "CS50".

# Let's look at what's actually happening in `renderHeader()` in `helpers.php`[11]:

```
/**
 * Renders header.
 */
function renderHeader($data = [])
{
    extract($data);
    require("header.php");
}
```

# `renderHeader()` takes an argument called `$data`, which has a **default value** of an empty array.

# The `extract` function takes our `$data` array and turns it into variables that we can use in `header.php`.

[11] http://cdn.cs50.net/2015/fall/lectures/8/m/src8m/mvc/2/helpers.php.src

- Here's our new version of `header.php`[12]:

  ```
  <!DOCTYPE html>

  <html>
      <head>
          <title><?= htmlspecialchars($title) ?></title>
      </head>
      <body>
          <h1><?= htmlspecialchars($title) ?></h1>
  ```

  - `<?= … ?>` is just syntactic sugar for `<?php echo(…) ?>`, where `echo` is merely a variant of `print`. This makes our print statements significantly more concise.
  - We just want to print the title, stored in `$title`, but we run it through the function `htmlspecialchars` to make sure it doesn't contain any characters or syntax that will break our site.
- If we look at the HTML generated by this PHP, it's identical to our first version of the site, but now we're not doing it all by hand!

• What other opportunities for design optimization do we have? Well, in `helpers.php`, we can see that `requireHeader()` and `requireFooter()` are identical, except that the former `require`s `header.php` and the latter `require`s `footer.php`. Instead, let's use a more general function, in version 3 of `helpers.php`[13]:

[12] http://cdn.cs50.net/2015/fall/lectures/8/m/src8m/mvc/2/header.php.src
[13] http://cdn.cs50.net/2015/fall/lectures/8/m/src8m/mvc/3/helpers.php.src

```php
<?php

    /**
     * Renders template.
     */
    function render($template, $data = [])
    {
        $path = $template . ".php";
        if (file_exists($path))
        {
            extract($data);
            require($path);
        }
    }

?>
```

# We can pass this function `"header"` or `"footer"` as its `$template` argument, where we think of a template as a blueprint for the HTML we want to output, but with some different values substituted in.

# We'll construct the path to the file by concatenating the argument passed as `$template` with the `.php` file ending. In PHP, we use a period (`.`) to concatenate strings - so much easier than concatenating strings in C, where we have to worry about allocating enough space in memory for the combined string.

# Now `index.php`[14] looks like this:

[14] http://cdn.cs50.net/2015/fall/lectures/8/m/src8m/mvc/3/index.php.src

```
<?php require("helpers.php"); ?>

<?php render("header", ["title" => "CS50"]); ?>

<ul>
    <li><a href="lectures.php">Lectures</a></li>
    <li><a href="http://cdn.cs50.net/2015/fall/lectures/0/w/syllabus/
cs50/cs50.html">Syllabus</a></li>
</ul>

<?php render("footer"); ?>
```

# A very similar `render` function will come in handy on Problem Set 7!

4. SQL

- So far, none of our web pages have been able to save any data provided by the user. The only time we've ever saved anything we did in this class was a little bit with file I/O in C, using `fopen`, `fprintf`, and the like, and writing to a CSV file.
  # A CSV file would be the very simplest form of a **database**, whereby we could save information provided by our users, like if we wanted to actually register students on our frosh IMs website.
- More generally, a database is a structure that lets you store and query data.
- We'll now introduce **SQL**, pronounced sequel, or **Structured Query Language**, a programming language for databases. A database can also be thought of like a fancy version of Excel or Apple Numbers, but unlike those programs, SQL lets us store and query huge amounts of data (many millions of rows!).
  # Until recently, Excel would only let you store up to 65535 rows of data (Microsoft was clearly using a 16-bit value to store the row number!)
- SQL is very powerful, but can really be boiled down to just four main operations: `DELETE`, `INSERT`, `UPDATE`, and `SELECT`.
  # `SELECT` lets you retrieve data from your database.
  # `INSERT` adds new rows to your database.
  # `DELETE` deletes rows from your database.

  - `UPDATE` lets you change part of an existing row.

- Preinstalled in the CS50 IDE is MySQL, a version of the SQL programming language that's widely used, and a web-based tool for interacting with it called phpMyAdmin (because it's written in PHP, not because we use PHP to work with it).
- On Problem Set 7, you'll be building a site called C$50 Finance, which will involve a database containing users (with ids, usernames, hashed passwords, and dollar amounts that they have in their bank accounts), so users can register and log in to your site. You'll also have a database to store the stocks that users have in their portfolios.
  - It's critical that we not store passwords as plaintext, instead hashing them (i.e., encrypting them) so if someone breaks into the database, they can't just take the passwords directly.
  - It's still possible to get the passwords from the encrypted versions, but it takes some significant brute-forcing using dictionary attacks and so on. Some hashes are more secure than others, and will take longer to crack.
- When we're storing values in MySQL, we do need to take into account datatypes (of which there are many!):
  - `CHAR`
  - `VARCHAR`
  - `INT`
  - `BIGINT`
  - `DECIMAL`
  - `DATETIME`
  - … and more.
- Deciding how to store values in a database is one of the challenges of database design.
- We'll talk more about things like indexing and searching next time.