Week 7, continued

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. PHP

- At this point, we're pretty much done with teaching you languages. Last lecture, we looked at HTML tags and attributes, and now we're going to look at PHP, but we're not going to teach you PHP per se - we'll run through it quickly and give you the tools to understand this language.

1.1. Syntax

- In PHP, there's no need for a main function - the code in your PHP file (that isn't wrapped in some other function) is just executed from the top.
- Variables in PHP are declared as follows:

  ```
  $s = "hello, world";
  ```

  # This looks pretty similar to C, but note that we're not declaring the variable's type, instead just prepending all variable names with `$`. PHP is **loosely typed**, whereas C is **strongly typed**, meaning that PHP knows the difference between datatypes but the computer will figure it out for you, whereas in C you have to tell the computer exactly what type each variable is.

- Conditions in PHP look identical to in C:

```
if (condition)
{
    // do this
}
else if (condition)
{
    // do that
}
else
{
    // do this other thing
}
```

# Also exactly as in C, boolean conditions can be combined with `&&` or `||`.

- Switches are also functionally equivalent to those in C:

```
switch (expression)
{
    case i:
        // do this
        break;

    case j:
        // do that
        break;

    default:
        // do this other thing
        break;
}
```

# However, in C we're limited to comparing only certain datatypes in a `switch` statement (`int`, `char`, etc), whereas in PHP we can compare full-fledged strings and the like without worrying about pointers.

- The loops we've seen before are structured the exact same way, too, such as the `for` loop:

```
for (initializations; condition; updates)
{
    // do this again and again
}
```

- But now we add an additional loop, called a `foreach` loop, that lets us iterate over an array without worrying about the details of indices:

```
foreach ($numbers as $number)
{
    // do this with $number
}
```

  # We’re assuming here that `$numbers` is an array containing a bunch of numbers. On each iteration of this loop, PHP will update `$number` to contain the value of the next element of `$numbers`.

- We can declare an array in PHP like this:

```
$numbers = [4, 8, 15, 16, 23, 42]
```

- Even more powerfully, PHP supports **associative arrays**, or arrays of key-value pairs. Rather than elements being indexed by some number between 0 and the length of the array, elements are indexed by their keys:

```
$quote = ["symbol" => "FB", "price" => "79.53"];
```

  # Each key, value pair is indicated by `"key" # "value"`.

  # Most other languages also support associative arrays in some way.

  # Whereas in C we might index into an array as `quote[i]`, and have to remember which value was in which numeric index, in PHP we can do `$quote["symbol"]` to directly access the symbol field.

- PHP does have a huge variety of other features, including many, many more functions already written for you than in C.
- In PHP, we no longer have to worry about low-level, powerful details of memory management (e.g., `malloc`, `free`, pointers more generally).
- Let’s write a super simple program in PHP as follows:

```php
<?php

    printf("hello, world\n");

?>
```

# Any PHP file needs to start with the syntax `<?php` and end with `?>` so the computer knows to interpret everything between these tags as PHP code.

# We don't need to run `make hello` on this, because PHP is not a compiled language like C - it's an **interpreted language**, meaning that it's not compiled into machine code, but rather passed as input to an **interpreter**, a program that understands PHP code line by line.

  # The interpreter essentially runs a big loop with many conditions that iterates over each line of code that the programmer wrote and executes the appropriate action corresponding to the line of code.

  # This means that there's no `make`, no object code, and no binary executable at the end - just our source code file. We can only run the source code file via the interpreter, since it's not directly in machine language (`0`s and `1`s) that the computer can understand itself.

# So to run our program, we actually pass our source code as input to the interpreter program, which is just called `php`:

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

- We don't even need to use `printf` in PHP - we can just use a function called `print`, and rather than using `printf`'s format strings to print variable values, we have simpler ways of doing that (although `printf` itself works pretty much identically in PHP as in C).
- PHP has a few syntactic differences from C (although overall it's quite similar), but the programming concepts at work are very much the same.

- Recall that a few weeks ago we had a code example called `conditions-1.c`[1] - let's translate it into PHP:

```
<?php

    // ask user for an integer
    $n = readline("I'd like an integer please: ");

    // analyze user's input
    if ($n > 0)
    {
        printf("You picked a positive number!\n");
    }
    else if ($n == 0)
    {
        printf("You picked zero!\n");
    }
    else
    {
        printf("You picked a negative number!\n");
    }

?>
```

  # Note that rather than `int n = GetInt();`, we have in line 4, `$n = readline("I'd like an integer please: ");` - where `readline` is a function that accepts a user's input, much like `GetInt()` and the like from the CS50 Library in C, but it takes as an argument the text to print as a prompt for the user.

  # Apart from that, this looks almost identical to our original C version, with really the only change being that our variables are all now prefixed with `$`.

  # But again, unlike in C, this isn't going to be compiled - instead, we run it using the `php` interpreter:

```
jharvard@ide50:~/workspace $ php conditions-1.php
I'd like an integer please: 50
You picked a positive number!
```

[1] http://cdn.cs50.net/2015/fall/lectures/1/w/src1w/conditions-1.c

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

- Let’s try something a little more powerful. In PHP, we can implement Problem Set 5 much more simply, as in `dictionary.php`[2]. We’ll assume `speller` has been implemented for us (in fact, it’s almost identical to the C version, line by line). We’ll start like this:

```
<?php

    $size = 0;

    $table = [];
...
```

  # As we did in C, we can store the size of our dictionary in a global variable.

  # Implementing a hash table in PHP is just `$table = [];`, because a hash table can be thought of as an associative array where the keys are the words and the values are `true` to indicate that the word is present in the dictionary.

- Continuing on, we can implement the `size` function much as we would in C:

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

  # In PHP, if we want to use a global variable called `$size` inside a function, we have to tell the function to look for that variable in the global namespace using `global $size`.

  # Note also that functions' return types in PHP don’t need to be specified (and functions are just declared as `function`).

- How about `load`?

[2] http://cdn.cs50.net/2015/fall/lectures/7/w/src7w/mispellings/dictionary.php.src

```
...
    function load($dictionary)
    {
        global $table, $size;
        if (!file_exists($dictionary) && is_readable($dictionary))
        {
            return false;
        }
        foreach (file($dictionary) as $word) ❶
        {
            $table[chop($word)] = true; ❷
            $size++;
        }
        return true;
    }
...
```

  # First notice in the line labeled **1** that we can read the lines of a file into an array with the PHP function `file`, which takes a filename as its argument and returns the array containing all the lines of the file. We don't have to handle any of the details of file I/O by hand.

  # In the line labeled **2**, we add the word in question to our hash table by indexing into our array, not by number but using the word itself as the key.

  # Note that these two lines implement the complete functionality of loading words from a file into a hash table - everything else in this function is error checking and tweaks.

- Let's continue with `check`:

```
...
    function check($word)
    {
        global $table;
        if (isset($table[strtolower($word)])) ❶
        {
            return true;
        }
        else
        {
            return false;
        }
    }
...
```

- # Since the only value we're ever storing in the associative array is just `true`, we can check whether a word is in our dictionary by just checking `isset($table[$word])`, i.e., does that key exist in our associative array, as in the line labeled **3** above.

- Let's finish this with `unload`:

```
...
    function unload()
    {
        return;
    }

?>
```

- # We never `malloc`ed anything, so there's nothing to `free`! We don't need to worry about the memory management details of our data structure, because PHP will handle them for us under the hood. This is a feature of higher-level languages that really streamlines our code.

- What's the price we're paying for our code being this much simpler? Well, let's compare this implementation's performance to the staff solution to Problem Set 5 on a large text like the King James Bible:

```
jharvard@ide50:~/workspace/src7w/mispellings $ ~cs50/pset5/speller texts/
kjv.txt
[... many misspelled words ...]

WORDS MISSPELLED:       33441
WORDS IN DICTIONARY:    143091
WORDS IN TEXT:          799460
TIME IN load:           0.03
TIME IN check:          0.52
TIME IN size:           0.00
TIME IN unload:         0.01
TIME IN TOTAL:          0.56

jharvard@ide50:~/workspace/src7w/mispellings $ php speller texts/kjv.txt
[... many misspelled words ...]

WORDS MISSPELLED:       33441
WORDS IN DICTIONARY:    143091
WORDS IN TEXT:          799460
TIME IN load:           0.19
TIME IN check:          1.07
TIME IN size:           0.00
TIME IN unload:         0.00
TIME IN TOTAL:          1.26

jharvard@ide50:~/workspace/src7w/mispellings $
```

- So the C version takes 0.56 seconds to spell-check the King James Bible, while the PHP version takes 1.26 seconds. Still pretty fast, but noticeably slower.
- Interpreted languages in general are going to be slower than compiled languages like C, because the interpreter has to read them line by line, and so the conversion to machine code is happening at the same time as the program running.
- So our implementation time is significantly less (5 minutes in PHP versus probably several hours in C), but we pay for it with longer running time (which might make more of a difference than the 0.7 seconds above if we were working with larger datasets or on older hardware).
  - Note also that although our programs of tens or hundreds of lines of C code compile very quickly, huge pieces of software like the commercial programs

produced by Microsoft or Google might contain many millions of lines of code, and could take much more time to compile (several seconds to minutes or even, for particularly enormous programs, hours).

1.2. Using PHP to Send Texts

- It's even possible to do much more impressive things, like send a text message to everyone in CS50 programmatically, using PHP.
  - This demo has not been uniformly successful in the various years that we've tried it.
  - We have everyone's names, phone numbers, and carriers stored in a **CSV**, or comma-separated values, file.
  - It's possible to send texts via email using SMS-to-email gateways - most carriers provide an email address that you can append to your number to send a text to your phone via email.
  - The basic structure of our code is going to start out looking something like this:

```
<?php

    $handle = fopen("students.csv", "r");
    if ($handle == false)
    {
        printf("could not open file\n");
        exit(1); ❶
    }

    $addresses = [];

    while ($row = fgetcsv($handle)) ❷
    {
        $number = $row[2];
        $carrier = $row[3];

        if ($carrier == "AT&T")
        {
            $address = "{$number}@txt.att.net"; ❸
            $addresses[] = $address; ❹
        }

        // repeat for several other carriers
    }

    fclose($handle);
...
```

# Note in the line labeled **1** that we don't `return 1;`, because we're not inside a function - instead, we `exit(1);` from the PHP interpreter to indicate that an error has occurred.

# The function `fgetcsv`, in the line labeled **2**, takes a file handle as its argument and returns an array containing the values in each column of the CSV for the next row (i.e., it parses the line by splitting it on commas).

# We don't need to mess around with format strings in PHP; in the line labeled **3** we can see that to construct an email address from someone's phone number, we can just insert it directly into the string, wrapping the variable name in curly braces as in `$address = "{$number}@txt.att.net";`.

  - In the line labeled **4**, we can see how to push an element onto the end of an array (equivalent syntax would be `array_push($addresses, $address)`. We don't need to worry about growing the size of the array - PHP handles that for us.

- Now let's do the work of actually sending emails to all these addresses we've constructed:

```
...
    // instantiate mailer
    $mail = new PHPMailer(); ❶

    // configure mailer
    // http://phpmailer.worxware.com/index.php?pg=methods
    // http://phpmailer.worxware.com/index.php?pg=properties
    // https://www.google.com/settings/u/0/security/lesssecureapps
    $mail->IsSMTP();
    $mail->Host = "smtp.gmail.com";
    $mail->Password = "TODO";
    $mail->Port = 587;
    $mail->SMTPAuth = true;
    $mail->SMTPDebug = 1;
    $mail->SMTPSecure = "tls";
    $mail->Username = "TODO";

    // set From:
    $mail->SetFrom("bot@cs50.net");

    // set body
    $mail->Body = "Miss you! love, CS50 Bot";

    // iterate over email addresses
    for ($i = 0, $n = count($addresses); $i < $n; $i++) ❷
    {
        // add email address to To: field
        $mail->addAddress($addresses[$i]); ❸

        // send email
        if ($mail->Send())
        {
            print("Sent text #{$i}.\n");
        }
        else
        {
            print($mail->ErrorInfo);
        }

        // clear To: field
        $mail->ClearAddresses(); ❹
    }

?>
```

    - In the line labeled **5**, we use a library called `PHPMailer` that gives us the ability to send emails. Below that, you can see various settings, including setting the `Host` to `smtp.gmail.com`.
    - Note in the line labeled **6** that we no longer need to keep around the size of an array - we can now get it by using the `count` function.
    - `$mail` here is essentially a `struct` (which the arrow notation should remind you of), which can contain not only values but also functions (as we can see in the line labeled **7**), called methods, because PHP is an **object-oriented** language.
    - The line labeled **8** is key! Without this, on each iteration, everyone who's gotten a text so far gets another text, because they would remain in the address list! (This actually happened when we attempted this demo in 2011.)
  - Volunteer Maya comes up to actually hit enter to send texts to everyone…
    - As usual, we have some technical difficulties, but instead we use our program to spit out everyone's text-to-email addresses and then paste them into Gmail.

2. Using PHP for Web Programming

- We're moving into PHP not so we can write more command-line programs more easily, but instead so we can start writing code that generates web pages.
  - We can construct in PHP, as David did many years ago in Perl, a website that lets users sign up for freshman intramural sports by taking input from the user and storing it in some way.
  - If we look at `froshims-0.php`[3], we can see that it's actually all HTML, except for a PHP comment at the top, because PHP was designed to be intermingled with HTML:

[3] http://cdn.cs50.net/2015/fall/lectures/7/w/src7w/froshims/froshims-0.php.src

```
<?php

    /**
     * froshims-0.php
     *
     * David J. Malan
     * malan@harvard.edu
     *
     * Implements a registration form for Frosh IMs.
     * Submits to register-0.php.
     */

?>

<!DOCTYPE html>

<html>
    <head>
        <title>Frosh IMs</title>
    </head>
    <body style="text-align: center;">
        <h1>Register for Frosh IMs</h1>
        <form action="register-0.php" method="post">
            Name: <input name="name" type="text"/>
            <br/>
            <input name="captain" type="checkbox"/> Captain?
            <br/>
            <input name="comfort" type="radio" value="less"/> Less
 Comfortable
            <input name="comfort" type="radio" value="more"/> More
 Comfortable
            <br/>
            Dorm:
            <select name="dorm">
                <option value=""></option>
                <option value="Apley Court">Apley Court</option>
                <option value="Canaday">Canaday</option>
                [... all the other freshman dorms ...]
            </select>
            <br/>
            <input type="submit" value="Register"/>
        </form>
    </body>
</html>
```

# This form submits to another file called `register-0.php`[4], which is also mostly HTML, but at least has one line of actual PHP code:

```
<?php

    /**
     * register-0.php
     *
     * David J. Malan
     * malan@harvard.edu
     *
     * Dumps contents of $_POST.
     */

?>

<!DOCTYPE html>

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

# `print_r` is a print function that's really just intended for debugging, but it shows that we can use the values passed through a form to change what's displayed on a page.

# Instead let's look at `froshims-2.php`[5], which is identical to `froshims-0.php`, except that instead of submitting to `register-0.php` it submits to `register-2.php`[6]:

[4] http://cdn.cs50.net/2015/fall/lectures/7/w/src7w/froshims/register-0.php.src
[5] http://cdn.cs50.net/2015/fall/lectures/7/w/src7w/froshims/froshims-2.php.src
[6] http://cdn.cs50.net/2015/fall/lectures/7/w/src7w/froshims/register-2.php.src

```
<!DOCTYPE html>

<html>
    <head>
        <title>Frosh IMs</title>
    </head>
    <body>
        <?php if (empty($_POST["name"]) || empty($_POST["comfort"])
 || empty($_POST["dorm"])): ?>
            You must provide your name, comfort, and dorm!  Go <a
 href="froshims-2.php">back</a>.
        <?php else: ?>
            You are registered!  (Well, not really.)
        <?php endif ?>
    </body>
</html>
```

# Although we're still not really registering anyone - i.e., we're not storing anyone's information anywhere - we can do some error checking here using PHP.

# We kind of glossed over this in the previous example, but `$_POST` is an example of a **superglobal** variable containing the values that were passed to your backend using the HTTP `POST` method. PHP turns the contents of the HTTP request, all those key-value pairs, into an associative array, so if you want to get at the name that a user submitted in your form, you can access it using `$_POST["name"]`.

- This was a bit of a whirlwind tour, but we'll continue with PHP next week!