Introduction to PHP

Part 1

Objectives

- Discuss the origin and use of server-side scripting using PHP
- Explain the syntax and basic constructs of PHP

PHP Origins and Uses

Overview of PHP

PHP is a server-side scripting language whose scripts are embedded in HTML documents

Similar to JavaScript, but on the server side

The PHP processor has two modes:
- copy (HTML)
- interpret (PHP)

PHP syntax is similar to that of JavaScript

Origins and Uses of PHP

PHP was originally developed by Rasmus Lerdorf in 1994
*   Developed to allow him to track visitors to his Web site

PHP is an open-source product

PHP was originally an acronym for Personal Home Page, but later it became ***PHP: Hypertext Preprocessor***

PHP is used for form handling, file processing, and database access

PHP Syntax and Constructs

General Syntactic Characteristics

PHP code can be specified in an HTML document internally or externally:

Internally:
```
<?php
  …
?>
```

Externally:
```
include ("myScript.inc")
```

A file can have both PHP and HTML

General Syntactic Characteristics

PHP has an extensive library of functions

A list of keywords:
   http://php.net/manual/en/reserved.keywords.php

Comments - three different kinds (Java and Perl)
```
//  . . .
#  . . .
/*  . . .  */
```

Statements end with semicolons ;

Compound statements are formed with braces { }

Identifiers

In PHP *identifiers* are used to name variables, functions, constants and classes

General rules:
- Identifiers can include letters, digits and the underscore
- First character must be either a letter or an underscore

Variable names
- Begin with a **dollar sign ($)**
- **Case-sensitive**

Function names are **not case-sensitive**

Class names are **not case-sensitive**

Primitive types

There are 8 primitive types:
- Four scalar types
  - **Integer**
  - **Double**
  - **Boolean**
  - **String**
- Two compound types
  - **Array**
  - **object**
- Two special types
  - **resource**
  - **NULL**

Scalar Types

Integer
- 4 bytes
- Can be written in decimal, octal or hexadecimal
- Use `is_int()` to test if a value is an integer

Double
- Can specified using standard notation or floating-point notation
- Use `is_float()` to test if a values is a floating-point number

Strings

String literals use single or double quotes

Single-quoted string literals
- Embedded variables are NOT interpolated
- Embedded escape sequences are NOT recognized

Double-quoted string literals
- Embedded variables ARE interpolated
- If there is a variable name in a double- quoted string but you don't want it interpolated, it must be backslashed
- Embedded escape sequences ARE recognized

Use `is_string()` to test whether a value is a string.

Boolean values

Boolean - values are `true` and `false`
- case insensitive

The following values are *false*
- `0, 0.0` and `""` and `"0"`
- Null
- Arrays with zero elements
- Objects with no values or functions

Everything else will evaluate to *true*

Use `is_bool` to test whether a values is a Boolean

Resource and NULL

Resources is anything that is not PHP data

- Holds a handle to the actual data
- Example: database, image

NULL represents that a variable has no value

Variables

Variables in PHP are identifiers prefaced with a $

There are **no type declarations**

There is **no explicit syntax for declaring variables**
- First time the value of variable is set the variable is declared

Variables may hold any type of data
- No compile-time or run-time checking
- Loosely typed

An unassigned (unbound) variable has the value, **NULL**

Variables (cont.)

The `unset` function sets a variable to `NULL`

The `isset` function is used to determine whether a variable is `NULL`

PHP has many ***predefined variables***, including the **environment variables** of the host operating system

You can get a list of the predefined variables by calling `phpinfo()` in a script

Variable variables

- You can reference the value of variable whose name is stored in another variable
- Use **$$**

Variable references

- Allow you to have two variables pointing to the same data
- Use **&** to create a reference

Constants

Constants are set using the *define statement*

Example:
```
define('NAME', "John");
define ('PI', 3.1415);
```

Once set cannot be changed

Output

Output from a PHP script is HTML that is sent to the browser

HTML is sent to the browser through **standard output**

We will use `echo` for basic output (book uses `print`)
  • **Echo is more efficient**

Example:
```
echo "This is too <br /> much fun <br />";
echo 72;
```

Output (cont.)

For formatted output use `printf`

Works like in C++

`printf`(*literal_string*, *param1*, *param2*, *...*)

PHP code is placed in the body of an HTML document

```
<html>
  <head><title> Trivial php example </title>
  </head>
  <body>
    <?php
    echo "Welcome to my Web site!";
    ?>
  </body>
</html>
```

EXAMPLE: today.php

Arithmetic Operators and Expressions

Usual operators: +, -, *, /, %

PHP operators and operator precedence:
http://php.net/manual/en/language.operators.php

Arithmetic functions:

`floor`, `ceil`, `round`, `abs`, `min`, `max`, `rand`, **etc.**

The only operator is period (**.**), for ***concatenation***

Indexing - `$str{3}` is the fourth character

String Functions
- `strlen, strcmp, strpos, substr,` as in C
- `chop` – remove whitespace from the right end
- `trim` – remove whitespace from both ends
- `ltrim` – remove whitespace from the left end
- `strtolower, strtoupper`

Automatic type conversion

PHP will automatically convert one type of variable to another whenever possible

Implicit conversion rules:
- Float and integer
    - integer is converted to a floating-point number
- Integer and string
    - String is converted to a number

Explicit conversions

Even though PHP is loosely typed there are occasions when it's useful to cast a value as a specific type.

- Casting operator is type inside parenthesis
  Ex: `(int)$total`
- Or can use functions
  `intval($total)` or
  `settype($total,"integer")`

The type of a variable can be determined with

- `gettype` or `is_type`
- `gettype($total)` - it may return `"unknown"`
- `is_integer($total)` – a predicate function

Relational operators

Relational operators compare numbers numerically and strings lexicographically
- If string is entirely numeric, a numeric comparison is made
- Otherwise a *lexicographic comparison* is made

Equality vs. identical
- **Equality (==)** compares if two values are equal
- **Identical (===)** compares if two values are the same type and are equal

Not-equal vs. Not identical
- **Not-equal (!= or <>)**
- **Not identical (!==)**

Logical operators

And: `&&, and`

Or: `||, or`

Exclusive or: `xor`

Not: `!`

Selection Statements

`if, if-else, elseif`

`switch` – like C++

The switch expression type must be **integer, double, or string**

Iteration

`while` - just like C

`do-while` - just like C

`for` - just like C

`foreach` - discussed later with arrays

Can use `break` - **in any** `for, foreach, while, do-while,` **or** `switch`

Can use `continue` - **in any loop**

EXAMPLE: powers.php

HTML can be intermingled with PHP script, e.g.:

```php
<?php
$a = 7;
$b = 7;
if ($a == $b) {
  $a = 3 * $a;
?>
<br /> At this point, $a and $b are equal <br />
So, we change $a to three times $a
<?php
  }
?>
```

Summary

- PHP is a server-side scripting language whose scripts are embedded in HTML
- Has two modes: copy and interpret
- The syntax is similar to C/C++ or JavaScript
- Scripts are included by <?php ... ?>
- Variables start with dollar sign ($)
- There are no explicit type declarations

PHP Arrays and Functions

Objectives

- Discuss how PHP arrays are implemented and used

- Explain the use and implementation of PHP functions

PHP Arrays

PHP Arrays

PHP arrays are implemented differently than in most other programming languages

A **PHP array** is really a mapping of keys to values, where the keys can be numbers (to get a traditional array) or strings (to get a hash)

Example: mapping from student name to G.P.A.

```
"John"   → 3.86
"Mary"   → 2.75
"Alice"  → 3.05
```

PHP Arrays

Two types:

- ***Indexed arrays***
  - Zero-subscripted
- ***Associative arrays***
  - **Keys are strings**
  - Like a two column table where the first column is the key and the second column is the value
  - All PHP arrays are stored internally as associative arrays

PHP arrays have an internal order, usually the order the elements were inserted

Storing data in arrays

Can use simple assignment to initialize an array

- **Indexed array**
  ```
  $addresses[0]="Lewis University";
  $addresses[1]="Computer Science";
  $addresses[2]=3.85;
  ```

- **Associative array**
  ```
  $addresses['univ']="Lewis University";
  $addresses['major']="Computer Science";
  $addresses['gpa']=3.85;
  ```

Note: **storing a value in an array will create the array
if it didn't already exist**
(but trying to retrieve a value from an array
that hasn't been defined won't create the array)

Can also use the **`array()`** construct, which takes one or more **key** => **value** pairs as parameters and returns an array of them

- The **keys** are non-negative integer literals or string literals
- The **values** can be anything
- Example:
  ```
  $list = array(0 => "apples",
          1 => "oranges", 2 => "grapes")
  ```

If a key is omitted and there have been integer keys, the default key will be the **largest current key + 1**

- Example:
  ```
  $days=array(1=> "Monday", "Tuesday",
  "Wednesday", "Thursday", "Friday", "Saturday")
  ```

Storing data in arrays

If a key is omitted and there have been no integer keys, `0` **is the default key**

If a key appears that has already appeared, the **new value will overwrite the old one**

To construct an empty array, pass no arguments to array(), **e.g.**: `$addresses = array();`

Arrays can have mixed kinds of elements, e.g.:

```php
$list = array("make" => "Cessna",
              "model" => "C210",
              "year" => 1960,
              3 => "sold");

$list = array(1, 3, 5, 7, 9);

$list = array(5, 3 => 7, 5 => 10,
              "month" => "May");

$colors = array('red', 'blue', 'green',
                'yellow');
```

Empty square-brackets (**[]**) can be used to insert values to the end of the existing array

Example:
```
$animals[]="dog";
$animals[]=array("dog", "cat")
```

You cannot use this construct with associative arrays

Accessing Array Elements

Access specific elements by using the array variable's name followed by the key (index) enclosed in square brackets

Examples:
```
$age["Fred"];
$age[2];
$list[4] = 7;
$list["day"] = "Tuesday";
$list[] = 17;
```

Note 1: If an element with the specified key
         does not exist, it is created

Note 2: If the array does not exist, the array is created

The **keys or values can be extracted from an array**, e.g.:

```
$highs = array("Mon" => 74, "Tue" => 70,
                    "Wed" => 67, "Thu" => 62,
                    "Fri" => 65);
$days = array_keys($highs);
$temps = array_values($highs);
```

Can **test whether an element exists** using `array_key_exists`, **e.g.:**

```
    if (array_key_exists("Wed", $highs)) ...
```

An **array can be deleted** with `unset`
```
unset($list);
unset($list[4]);   #Deletes index 4 element
```

Some Array Functions

`is_array($list)` returns `true` if `$list` is an array

`in_array(17, $list)` returns `true` if `17` is an element of `$list`

`sizeof(`*an_array*`)` returns the number of elements
- Also `count()`

`explode(" ", $str)` creates an array with the values of the words from `$str`, split on a space

`implode(" ", $list)` creates a string of the elements from `$list`, separated by a space

Extracting multiple values from an array

To **copy all of an array's values into variables**, use the **list()** function

Example:
```
$person=array("Cosmo", 38, "Jerry");
list($name, $age, $friend) = $person;
```

If you have more values in the array than in the `list()`, extra values are ignored

If you have more values in the `list()` than in the array, extra values are set to `NULL`

To extract a subset of the array,
use the `array_slice()` function

It returns a new array consisting of a consecutive series of values from the original array

Arguments
- Array: name of the array being sliced
- Offset: initial element in the slice
- Length: number of values to copy

Example:
```
$simpsons = array("Homer", "Marge", "Bart", "Lisa",
"Maggie", "Grandpa");
$simpsons_kids = array_slice($simpsons, 2, 3);
```

`simpsons_kids` **now contains** `["Bart", "Lisa", "Maggie"]`

Traversing arrays

The most common task with arrays is to do something to every element

There are several different ways to traverse arrays in PHP:

- The `foreach` construct

- The iterator functions

- Using a `for` loop

foreach

The most common way to loop over elements in array is to use the **`foreach`** **construct**

Elements are processed in their internal order

The construct **operates on a copy of the array** so changes made during iteration are not reflected

Syntax:
```
foreach($addresses as $addr)
        echo $addr . "<br>";
```

An alternative form gives you access to the current key, e.g.:
```
foreach($addresses as $key => $value)
    echo "$key  address is $value <br>";
```

Iterator functions

An alternative way of accessing arrays is using iterators

Every PHP array keeps track of the current element that is being accessed

The pointer to this element is called an *iterator*

PHP has functions to set, move and reset the iterator

Iterator functions:

- `current()` – Returns the element currently pointed at by the iterator
- `reset()` – Moves the iterator to the first element in the array and returns it
- `next()` – Moves the iterator to the next element in the array and returns it
- `prev()` – Moves the iterator to the previous element and returns it
- `each()` – Returns the key and value of the current element as an array and moves the iterator to the next element in the array
- `key()` – Returns the key of the current element

The **each()** function can be used to loop over the elements of an array

Example:

```
$colors = array("red", "yellow", "green", "purple", "blue");
reset($colors);
while ($element = each($colors))
    echo($element['value']. "<br />");
```

Note: this function does not make a copy of the array when traversing

for loop

If you are working with an indexed array where indexes are consecutive integers beginning at zero you can use a for loop

Example:
```
$colors = array("red", "yellow", "green", "purple", "blue");
for($i=0; $i < count($colors); $i++){
    echo $colors[$i] . "<br>";
}
```

Searching for values

The **in_array()** function returns true or false, depending on whether the first argument is an element in the array (second argument)

PHP indexes the values in arrays so `in_array()` is much faster than looping through the array and checking every value

The **array_search()** function takes the same arguments but returns the key of the value instead of true or false

Sorting

PHP provides three ways to sort arrays:
1. Sorting by keys
2. Sorting by values without changing keys
3. Sorting by values and then changing keys

The following functions sort by values
and reassign the keys starting at 0 :
- `sort()` for ascending order
- `rsort()` for descending order
- `usort` for user-defined order

These functions are designed to work on indexed arrays

Example:

```
$list = ('h', 100, 'c', 20, 'a');
sort($list);
// Produces ('a', 'c', 'h', 20, 100)
```

The following functions sort by values
but leave keys alone:
- `asort()` for ascending order
- `arsort()` for descending order
- `ausort()` for user-defined order

The sorted order can only be accessed by using traversal functions such as `foreach` and `next`

Sorting by keys

The following functions sort by key
- `ksort()` for ascending order
- `krsort()` for descending order
- `uksort()` for user-defined order

Example:
```
$list("Fred" => 17, "Mary" => 21,
      "Bob" => 49, "Jill" => 28);
ksort($list);
// $list is now ("Bob" => 49,
// "Fred" => 17, "Jill" => 28, "Mary" => 21)
```

The sorted order can only be accessed by using traversal functions such as `foreach` and `next`

User-defined sorting

Requires that you provide a function that takes two values and returns a value that specifies the order of the two values in the sorted array

Return values should be
  1 : if first > second
 -1 : if first < second
  0 : if first = second

Natural sorting order

PHP's built-in sort functions correctly sort strings and numbers but they don't correctly sort strings that contain numbers

To sort strings that contain numbers use
- natsort()
- natcasesort()

Reversing arrays

The **`array_reverse()`** function reverses the internal order of elements in an array

Numeric keys are renumbered starting at `0`, string keys are not changed

Swapping keys and values

`The` **`array_flip()`** function returns an array that reverses the order of each original element's key-value pairs

Works best when the original array has unique values

Randomizing order

To traverse the elements in an array in random order, use the **`shuffle()`** function

All existing keys are replaced with consecutive integers starting at 0

PHP Functions

Functions

A *function* is a named block of code that performs a specific task (possibly given some input parameters)

Functions improve readability and reliability
- The code only needs to written a single time

Functions in PHP can either be predefined (part of the language) or user-defined

Calling a function

Syntax:
$some_value = function_name(param1, param2, …)

Examples

```
$length = strlen("PHP")    //sets $length to 3
$result = abs(ceil(-9.2)) //sets $result to 9
```

PHP functions

A complete list of functions can be found at:
http://php.net/quickref.php

Some interesting functions:

- Math functions
- Date functions
- String functions

Math functions

```
abs(number num)
ceil(float num)
cos(float num)
bindec(string binary_number) and decbin(int num)
hexdec(string hex_number) and dechex(int num)
floor(float num)
pow(number base, number exponent)
rand(int min, int max)
round(float num, int precision)
sin(float num)
sqrt(float num)
```

Date functions

date(string *date_format*, int *timestamp*)
- Check out the online reference for details
  http://php.net/manual/en/function.date.php
- Example:
  ```
  $time = date("H:I");
  //$time is current time in hours and minutes
  ```
- Example $today = date("F j, Y");

mktime(int *hour*, int *min*, int *second*, int *month*, int *day*, int *year*, int *is_dst*)
- Everything after hours is optional, but must provide some value
- Example:
  ```
  $pearl = mktime(0, 0, 0, 12, 7, 1941);
  echo "<p>Pearl Harbor was attacked on " .
      date("m/d/y", $pearl) . "</p>";
  ```

Date functions (cont.)

strtotime`(string` *time*`, int` *now*`)`
- Now is optional
- Uses American dates
- Returns a timestamp on success, FALSE otherwise
- Examples:
  ```
  strtotime("now");
  strtotime("10 September 2000");
  strtotime("next Thursday");
  strtotime("last Monday");
  ```

time`()`
- Returns the current time

Length: `strlen(`*str*`);`

Removing whitespace
- `chop(`*str*`)` – remove whitespace from the right end
- `trim(`*str*`)` – remove whitespace from both ends
- `ltrim(`*str*`)` – remove whitespace from the left end

Changing case
- `strtolower(`*str*`)` – converts *str* to lowercase
- `strtoupper(`*str*`)` – converts *str* to uppercase
- `ucfirst(`*str*`)` – capitalizes the first character of *str*
- `ucwords(`*str*`)` – capitalizes the first character of each word in *str*

HTML

htmlentities(*str*)
- Converts all characters with HTML entity equivalents into those equivalents

htmlspecialchars(*str*)
- Converts HTML special characters into entity equivalents

strip_tags(*str*, *tags_to_preserve*)
- Removes HTML tags from a string
- Optional second argument specifies tags to leave in the string
- Example:
```
$input="<strong>The <em>bold</em> tags
stay<strong>";
$output = strip_tags($input, "<strong>");
```

Comparing strings

== casts non-string operators to strings
    `3` and `"3"` are equal

=== does not cast
    `3` and `"3"` are not equal

Other relational operators work
- If comparing numbers and strings, strings are cast to the number `0` unless the string starts with a number
- Example:
```
$str="PHP";
$num = 8;
if($str < $num)
  echo ("$str < $num");
```

Comparing strings (cont.)

strcmp(*str1*, *str2*)
- returns a number less than `0` if *str1 < str2*
- returns a number greater than `0` if *str1 > str2*
- returns `0` if *str1 = str2*

strcasecmp(*str1*, *str2*)
- works like `strcmp` but converts the strings to lowercase first

Substrings

substr(*str*, *start*, *length*)
- *length* is optional
- otherwise goes from start to end of string

Other substring functions
- `substr_count` - counts the number of substring occurrences
- `substr_replace` - replaces text within a portion of a string

Miscellaneous string functions

strrev(*str*)
- Returns a reversed copy of *str*

str_repeat(*str*, *count*)
- Returns a string that repeats *str* count times

str_pad(*str*, *length*, *with*, *pad_type*)
- Returns *str* padded to *length* using *with*
- *with* is optional; the default is space
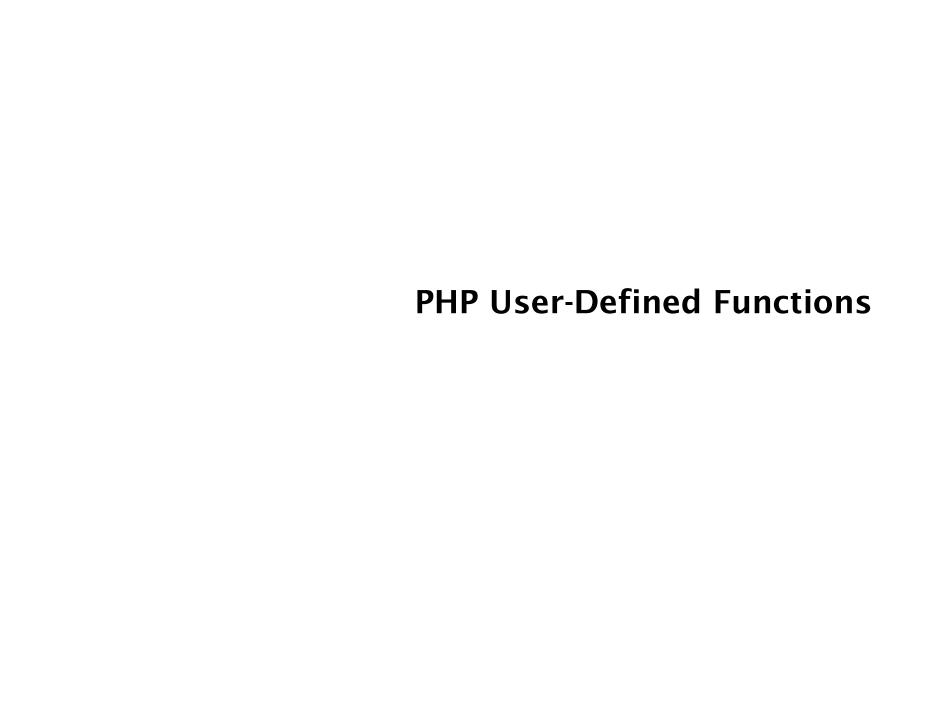- *pad_type* is optional and can be `STR_PAD_RIGHT`, `STR_PAD_LEFT` or `STR_PAD_BOTH`

Decomposing a string

explode (*separator*, *str*)
- Returns an array of strings

implode (*separator*, *array*)
- Returns a string created from other strings

PHP User-Defined Functions

Syntax:

```
function function_name(formal_parameters) {
   …
}
```

The function names can be any string that starts with a letter or underscore followed by zero or more letters, underscores and digits

General Characteristics of Functions

Functions need not be defined before they are called

Function **overloading is not supported**
- If you try to redefine a function, it is an error

Functions can have a variable number of parameters

Default parameter values are supported

General Characteristics of Functions

Function **definitions can be nested**

Function names are **NOT** case sensitive

The `return` statement is used to return a value
- If there is no `return`, there is no returned value

Function example

```php
function strcat($left, $right){
  $combined_string = $left . $right;
  return $combined_string;
}

$first= "Today is ";
$second = "Tuesday";

echo strcat($first, $second);
```

Variable scope

Variables defined in a functions, including parameters, are not available outside of the function

By default, variables defined outside a function are not available inside a function

Example:
```
$a= 3;
function foo() {
    $a += 2;
}
foo();
echo($a);
```

Global variables

To use a variable in the global scope within a function, use the keyword **global**

Example:

```
$a= 3;
function foo() {
    global $a;
    $a += 2;
}
foo();
echo($a);
```

Lifetime of Variables

Normally, the lifetime of a variable in a function is from its first appearance to the end of the function's execution

Can define *static variables*, that **retain their state between function calls**, e.g.:

```
static $sum = 0;   # $sum is static
```

Parameters

By default, parameters are **passed by value**

To **pass by reference**, precede the parameter name with an ampersand, e.g.:

```
function set_max(&$max, $first, $second) {
    if ($first >= $second)
        $max = $first;
    else
        $max = $second;
}
```

If the caller sends too many actual parameters, the subprogram ignores the extra ones

If the caller does not send enough parameters, the unmatched formal parameters are unbound

To **specify a default parameter**, assign the parameter value in the function declaration
- The assigned value **must be a constant**
- A function can have any number of parameters with default values
- They must be listed after all parameters that do not have default values

Variable parameters

A function may require a variable number of arguments

To do this leave out the parameter block and then use one of three functions to retrieve the parameters:

```
$array = func_get_args();


$count = func_num_args();


$value = func_get_arg(argument_number);
```

Example:
```
function count_list(){
    if(func_num_args()==0){
        return false;
    } else {
        $count = 0;
        for($i=0; $i<func_num_args(); $i++)
            $count++;
    }
    return $count;
}

echo count_list(1, 3, 5);
```

Return values

PHP can only return a single value with the keyword **`return`** (return multiple values using an array)

Any type may be returned, including objects and arrays, using `return`

By default, values are copied out of the function

To return a reference precede the function name with &

Example:
```
function & foo($n) ...
```
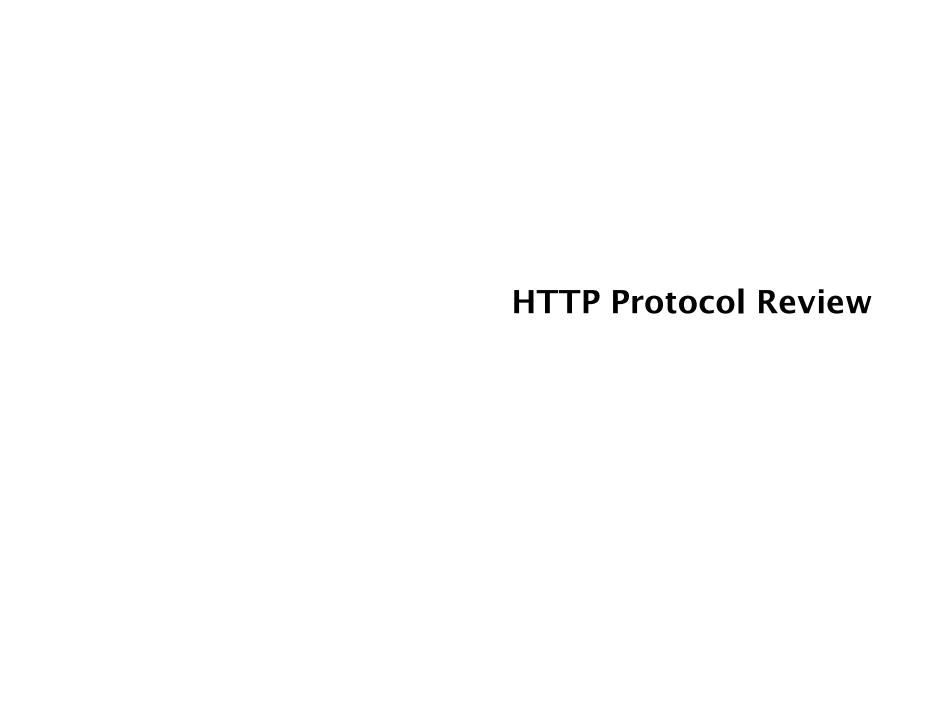
Summary

- PHP arrays are implemented as mapping of keys to values
- Arrays can be indexed by numeric keys or string keys
- Arrays can be traversed using foreach, iterators, or for loops
- There exist many functions for searching and sorting arrays
- PHP supports a range of functions for a variety of tasks, including dealing with dates, strings, HTML code, and others
- Function parameters can be passed and returned by value or by reference

PHP for Server Side Scripting

Part 3

Objectives

- Review the HTTP protocol
- How to access HTTP message data from PHP
- How to access and write files within PHP
- Explain solutions for maintaining state

HTTP Protocol Review

HTTP Protocol Review

When a web browser requests a web page, it sends an HTTP request message to a web server.

Message includes:

- Header information, e.g.:
  ```
  GET /index.html HTTP/1.1
  ```

- Optional header information
  ```
  User-Agent:  Mozilla/5.0 (Windows 2000; U) Opera 6.0 [en]
  Accept: image/gif, image/jpg,  text/*, */*
  ```

- Optionally a body

The web server then receives the request, processes it and sends a response

- First line is the status, e.g.:

```
HTTP/1.1 200 OK
```

- Additional headers, e.g.:

```
Date:   Sat 22 Jan 2006 20:25:12 GMT
Server:  Apache 1.2.22 (Unix) mod_perl/1.26 PHP/5.0.4
Content-Type:  text/html
Content-Length: 141
```

HTTP Methods

The two most common methods are `GET` and `POST`

GET is designed for retrieving information from the server
- A `GET` request encodes gathered information as part of the URL
- Users can bookmark `GET` requests
- `Get` requests can pass a limited amount of data

POST is meant for posting information to the server
- Actually is used for retrieving information (like GET)
- A `POST` request passes gathered information in the body of the HTML request
- Users cannot bookmark `POST` requests

A HTTP get Request

method

request

```
GET http://www.google.com/search?q=php HTTP/1.1
```

Sent as part of the URL:

- `Search` is the name of Google's server-side form handler
- ? Is a query string
  - Name/value pair
  - Multiple search strings are separated by **&**

Post Requests

A POST request is different from a GET request in the following ways:

- There's a block of data sent with the request, in the message body.

- There are usually extra headers to describe this message body, like **Content-Type:** and **Content-Length:**

- The *request* URI is not a resource to retrieve; it's usually a script to handle the data you're sending.

- The HTTP response is normally script output, not a static file.

A Post Request

The most common use of POST, by far, is to submit HTML form data to scripts.

Example:

method   request   protocol

```
POST /path/script.cgi HTTP/1.0
From: frog@jmarshall.com User-Agent: HTTPTool/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 32
home=Cosby&favorite+flavor=flies
```

header

data

Accessing HTTP Message Data from PHP

Superglobal Variables

Superglobals are built-in variables that are always available in all scopes

Collectively this information is referred to as **EGPCS** (environment, GET, POST, cookies and server):

- $_COOKIE
- $_GET
- $_POST
- $_FILES
- $_SERVER
- $ENV

Processing Forms

Use the **$_POST**, **$_GET** and **$_FILES** arrays to access form parameters from your PHP code
- Keys are parameter names
- Values are the values of those parameters

Usually copy the values from the array to a variable for processing, e.g.:
```
$title = $_POST['title'];
```

The type of method used to request a PHP page is available through `$_SERVER['REQUEST_METHOD']`

Automatic quoting of parameters

PHP ships with the `magic_quotes_GPC` option enabled

- This instructs PHP to automatically call **`addslashes()`** on all cookie data and `GET` and `POST` parameters
- Makes it easy to use form parameters in database queries
- Causes a problem with other uses because all single quotes, double quotes and backslashes are escaped with backslashes

- To get rid of the slashes uses the function **`stripslashes()`**, e.g.:
  ```
  $name = stripslashes($_POST['name']);
  ```

Handling newlines

A user can enter text over multiple lines in a form element such as a text area

To create the equivalent of these newlines in a webpage use the **nl2br()** function, e.g.:

```
$comments = nl2br($_POST['comments']);
```

Multi-valued parameters

HTML selection lists and checkboxes can allow multiple selections

To ensure that PHP recognizes the multiple values that the browser passes to a form-processing script, make the name of the field in HTML end with []

Example:
```
<select name="languages[]">
    <input name = "c">C</input>
    <input name = "c++">C++</input>
    <input name = "php">PHP</input>
    <input name = "perl">Perl</input>
</select>
```

When the user submits the form, **$_POST['languages']** contains an array instead of a string

Form Validation

Need to validate user data before storing or using it!

Can use JavaScript, but…
- User can disable JavaScript
- Browser might not support it

PHP provides a more secure way to do the validation

Some important validation functions/operators

empty() : tests whether a value was provided

is_numeric() : tests whether a value is a number

relational operators: !=, ==, >, <

strpos(haystack, needle) : returns the position of needle in haystack or false if needle is not found

preg_match(regex, str [,array]) : for testing regular expressions

Validating with PHP

Similar to validating with JavaScript, i.e.:

provide the user with error messages

Output buffering

In a normal PHP script, **any echo statement is sent to the browser as soon as it is executed**

With ***output buffering*** the HTML and data will instead be put into a buffer

Turn on output buffering with the `ob_start()` function
- All echos will send data into a buffer instead of to the browser
- HTTP calls (like `header()` and `setcookie()`) won't be buffered

Output buffering (cont.)

To discard the data in the buffer use the
`ob_end_clean()` function

To send the buffer to the browser use the
`ob_end_flush()` function

PHP Files

PHP Files

PHP can:
- Deal with any files on the server
- Deal with any files on the Internet, using either http or ftp

PHP associates a variable with a file, called the *file variable*

A file has a file pointer (where to read or write), e.g.:
`$fptr = fopen(`filename, use_indicator`)`

Use indicators: `r, r+, w, w+, a, a+`

Since `fopen` could fail, use it with `die`, e.g.:

```
$file=fopen("welcome.txt","r")
       or die("Unable to open file!");
```

Use `file_exists(`*filename*`)` to determine whether file exists before trying to open it

Use `fclose(`*file_var*`)` to close a file

To read all or part of the file into a string variable, use **fread**, e.g.:

`$str = fread(`*file_var*`, `*#bytes*`)`

(note: to read the whole file, use `filesize(`*file_name*`)` as the second parameter)

To read all of the lines of the file into an array, use **file**, e.g.:

`$file_lines = file(`file_name`)`

(need not open or close the file)

To read one line from the file use **fgets**, e.g.:

`$line = fgets(`*file_var*`, `*#bytes*`)`

- It reads characters until `eoln`, `eof`, or *#bytes*- chars been read

To read one character at a time use **fgetc**, e.g.:

`$ch = fgetc(`*file_var*`)`

You can control reading lines or characters with eof detection using **feof** (TRUE for eof; FALSE otherwise), e.g.:

```
 while(!feof($file_var)) {
     $ch = fgetc($file_var);
 }
```

Writing Files

To write to files, use **fwrite**, e.g.:

`$bytes_written = fwrite(`*file_va*r`, `*string*`)`

`fwrite` returns the number of bytes it wrote

Files can be locked (to avoid interference from concurrent accesses) with **flock**

Maintaining State

Maintaining State

HTTP is a ***stateless protocol***

Once a web server completes a client's request the **connection goes away**

No way for a server to recognize that a sequence of requests all originates from the same client

Why would we want to maintain state?

PHP has two main methods for tracking data:
- Cookies
- Sessions

Cookies

A *cookie* is a name/value pair that is passed between a browser and a server in the HTTP header

In PHP, cookies are created with **setcookie**
setcookie(*cookie_name*, *cookie_value*, *lifetime*)

Example:
```
setcookie("voted", "true", time() + 86400);
```

Cookies are implicitly deleted when their lifetimes are over

Cookies (cont.)

Few notes about cookies:

- Cookies are stored in a Web browser but only the site that originally sent a cookie can read it

- Cookies are read by a site when the page on that site is requested by the web browser

- Cookies are generally limited to about 4kb of total data

- Cookies **must be created before** any other HTML is created by the script

- Cookies are obtained in a script the same way form values are gotten, using the `$_COOKIES` array

Accessing cookies

When the browser sends a cookie back to the server, you can access the cookie through `$_COOKIE` array.

- Key is the cookie name
- Value is the cookie's value field

A cookie is **never accessible immediately** after it's been set

The page must be requested or reloaded for the cookie to be available

Deleting cookies

You can use the `setcookie()` function to delete a cookie

- It requires only one value, name
- Set the remaining parameters to blank or for expiration, a time in the past
- Example:
  `setcookie('user', '', time()-3600);`

Sessions

Sessions provide a way to track data for a user over a series of pages

When you start a session, PHP generates a random session ID -- a reference to that particular session and its stored data

Session ID is sent to the Web browser as a cookie

Subsequent PHP pages will use this cookie to retrieve the session ID and access the session information

Creating a session

Create a session using the `session_start()` function

This function sends a cookie so it **must be called prior to any HTML or white space**

The first time a session is started a random session ID is generated and a cookie is sent to the Web browser with the name `PHPSESSID` and a value

Record data by assigning values to the `$_SESSION` array, e.g.:

```
$_SESSION['name']='John';
```

Each time you do this, PHP writes the data to a temporary file stored on the server

Accessing session variables

Begin the session by calling the `session_start()` function

Access values by retrieving them
from the `$_SESSION` array

Session Tracking Example

Example using sessions - counting number of pages visited

```
session_start();
if (!isset($_SESSION['count'])) {
  $_SESSION['count'] = 0;
} else {
  $_SESSION['count']++;
}
$count = $_SESSION['count'];
echo "Visited $count times<br>";
```

Deleting a session

To delete a session, start with the `session_start()` function

Then delete the session variables by unsetting the `$_SESSION` array, e.g.:

```
unset($_SESSION);
```

Then remove the session data from the server, e.g.:

```
session_destroy();
```

Other session functions

Some other useful functions:

- Function **session_is_registered** returns true if the given variable is registered

- Function **session_id()** returns the current session ID.

Session life

By default **PHP session ID cookies expire when the browser closes**

Sessions don't persist after the browser ceases to exist

You can use cookies to allow some session information to persist after the browser is closed

Benefits of sessions over cookies

- **Session are generally more secure** because the data isn't repeatedly transmitted back and forth between the client and the server

- **Sessions let you store more information** than you can in a cookie

- **Session can be made to work even if the user doesn't accept cookies in their browser**

Benefits of cookies over sessions

- Marginally **easier to create and retrieve**

- Require slightly **less work from the server**

- Normally **persist over a longer period of time**

Use cookies in situations where:

- Security is less of an issue
- A minimum amount of data is being stored

Otherwise use sessions

- May require more effort when writing scripts

Summary

- HTTP protocol is used to send data between client and server using either GET or POST methods
- In PHP, form data from client can be accessed using superglobals $_GET and $_POST
- PHP needs to validate all input received before storing or using it
- Several PHP functions can be used to read and write from/to files: fread, fwrite, file, fgets, fgetc, etc.
- Maintaining state can be implemented using cookies or sesssions
- Cookies are easier to implement, but sessions are more secure and allow for storing more data