Quick Introduction to PHP

Dr. Chris Bourke
Department of Computer Science & Engineering
University of Nebraska—Lincoln
Lincoln, NE 68588, USA
Email: cbourke@cse.unl.edu

2016/01/22 11:49:31

Abstract

These are lecture notes used in CSCE 155 and CSCE 156 (Computer Science I & II) at the University of Nebraska—Lincoln. They represent a short introduction to the PHP programming language for students who already have a strong foundation in at least one high-level programming language.

Contents

1 Overview 2
   1.1 History .......................................................... 2
   1.2 Key Aspects ................................................... 3
   1.3 Hello World Demo ............................................. 3

2 Variables 4
   2.1 Using Variables ................................................. 5
   2.2 Type Juggling .................................................... 6

3 Operators 7

4 Strings 9

5 Arrays 11
List of Code Samples

1 Overview


1.1 History

- Developed by Rasmus Lerdorf (1995)
- Began as a collection of C modules that processed HTML and dynamically replaced special tags interleaved in static HTML
- Some aspects are due to this connection to C/HTML (case-insensitivity, lack of function overloading, c-style syntax, etc.)
- Pages were served through Common Gateway Interface (CGI)
- Version History
  - 1.0.0 1995 (Personal Home Page Tools)
  - 2.0.0 (1997)
  - 3.0.0 (1998)
1.2 Key Aspects

- Interpreted, not compiled; all errors are runtime errors
- C-style syntax: semicolons, bracketed code blocks, identifiers
- Supports OOP (added in PHP 3)
- Weakly typed (dynamically typed): variables are not declared; types are implicit
- No memory management (automated garbage collection)
- Main purpose is for dynamic web pages (you can interleave HTML and PHP)

1.3 Hello World Demo

- PHP code sections are denoted with 

- C-style comments
- Standard output: `print` (or `printf()`, `echo`)
- Command Line Interface (CLI) demo:

- Interleaving of HTML, dynamic HTML generation
- Web browser demo:
  - On CSE, anything you put into your `public_html` directory is accessible via `http://cse.unl.edu/~cselogin`
  - Permissions must be set appropriately (world readable)
  - Requests are handled by the webserver which invokes the PHP interpreter and serves the response
  - Response is rendered by your browser
  - Contrast: whitespace in CLI vs browser
<?php

/* A basic hello world program in PHP */

print "Hello World!\n";

?>

<html>
<head>
<title>Hello World PHP Program</title>
</head>
<body>
<?php

print "<h1>Greetings!</h1><h2>
"
;

print "<p>Hello World!</p>
"

?>

</body>
</html>

2 Variables

- Built-in types:
  - Booleans (bool, can be true or false)
  - Integers
  - Doubles (floats)
  - Strings
  - Arrays
  - Objects

- Standard assignment operator, =

- Standard numerical literals

- String and character literals: single or double quotes 'hello', "world"

- Keyword: null
– Case insensitive constant
– Treated as zero when used with numeric types
– Treated as empty string when used with string types

2.1 Using Variables

PHP is *weakly typed* (aka dynamically typed): variables do not have one fixed, explicit type. Instead, the type of a variable is implicit in the value it holds and may change depending on the value that gets assigned to it.

- Identifier rules: may use A-Z, a-z, 0-9, _ (plus the extended ASCII set); case sensitive
- Variables *must* begin with a dollar sign
- Constants may be defined
  - Case sensitive
  - Defining: `define('CONST', 'VALUE');`
  - When using constants, dollar sign is not used
  - Attempts to reassign constants result in a warning and have no effect

```php
1 $a = 10; // a is an integer
2 $a = "ten"; // now a is a string!
3
4 // constants
5 // defining:
6 define('PI', 3.1415);
7 define('COURSE', 'CSCE156');
8
9 // usage (no dollar sign):
10 $area = PI * PI * $radius;
11 print "This is " . COURSE;
12
13 // cannot redefine constants:
14 define('PI', 3); // warning, no effect
```
2.2 Type Juggling

- When types are mixed with certain operators, they are juggled to make the operations compatible
- Numeric types, when used with string concatenation, are converted to string representations
- Strings, when used with arithmetic operators are coerced into integers or floating point numbers using the leading numeric characters; if the conversion fails, the result is zero
- Better to use explicit conversion functions to guarantee types:
  - $a = intval("123");$
  - $b = floatval("3.14");$

- Checker functions can determine the types of strings:
  - `is_numeric("10")` (true)
  - `is_numeric("10h")` (false)
  - `is_int()`, `is_float()`

```php
1 //all of the following result in a being an integer with value 10
2 $a = intval("10");
3 $a = intval("10.0");
4 $a = intval(10.0);
5 $a = intval(10.5);
6 $a = intval("10.5foo");
7
8 //the following result in doubles with value 123.5
9 $b = floatval(123.5);
10 $b = floatval("123.5");
11 $b = floatval("123.5bar");
```
3 Operators

- Standard binary and unary operators: `+`, `-`, `*`, `/`, `%`, `++`, `--`

- String concatenation: period
  ```
  $msg = "hello" . " " . "world!";
  ```

- Alternatively, variables can be used directly inside static strings:
  ```
  $msg = "a has the value $a\n";
  ```

- Numeric relational operators: `<`, `<=`, `>`, `>=`

- Equality operators: since PHP is weakly typed we need a way to compare both the content and type of variables
  - Content equality (loose comparison): `==`, `!=`
  - Content and type equality (strict comparison): `===`, `!==`

- Logical operators: `!`, `&&`, `||`

- Checker functions:
  - `is_null($a)` — returns `true` if `$a` is not set or is set but is `null`
  - `isset($a)` — returns `true` only if `$a` is set and it is not `null`
- `empty($a)` – returns true if `$a` is an empty string, false, an empty array, `null`, "0", 0, or an unset variable
- A full listing of possibilities is in the table below

- **Bad algebra operations**
  - Division by zero: `$a = 1 / 0;` results in a *warning*, result is `null`
  - `sqrt(-1)` result is NaN
  - `log(0)` results in `-INF`
  - Checkers:
    - `is_nan($a)`
    - `is_infinite($a)`
<table>
<thead>
<tr>
<th>Value of $var</th>
<th>isset($var)</th>
<th>empty($var)</th>
<th>is_null($var)</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>bool(true)</td>
<td>bool(false)</td>
<td>bool(false)</td>
</tr>
<tr>
<td>&quot; &quot; (an empty string)</td>
<td>bool(true)</td>
<td>bool(true)</td>
<td>bool(false)</td>
</tr>
<tr>
<td>&quot; &quot; (space)</td>
<td>bool(true)</td>
<td>bool(false)</td>
<td>bool(false)</td>
</tr>
<tr>
<td>false</td>
<td>bool(true)</td>
<td>bool(true)</td>
<td>bool(false)</td>
</tr>
<tr>
<td>true</td>
<td>bool(true)</td>
<td>bool(true)</td>
<td>bool(false)</td>
</tr>
<tr>
<td>array() (an empty array)</td>
<td>bool(true)</td>
<td>bool(true)</td>
<td>bool(false)</td>
</tr>
<tr>
<td>null</td>
<td>bool(false)</td>
<td>bool(true)</td>
<td>bool(true)</td>
</tr>
<tr>
<td>&quot;0&quot; (0 as a string)</td>
<td>bool(true)</td>
<td>bool(true)</td>
<td>bool(false)</td>
</tr>
<tr>
<td>0 (0 as an integer)</td>
<td>bool(true)</td>
<td>bool(true)</td>
<td>bool(false)</td>
</tr>
<tr>
<td>0.0 (0 as a float)</td>
<td>bool(true)</td>
<td>bool(true)</td>
<td>bool(false)</td>
</tr>
<tr>
<td>var $var; (declared with no value)</td>
<td>bool(false)</td>
<td>bool(true)</td>
<td>bool(true)</td>
</tr>
<tr>
<td>NULL byte (&quot;\0&quot;)</td>
<td>bool(true)</td>
<td>bool(false)</td>
<td>bool(false)</td>
</tr>
</tbody>
</table>

Table 1: Results for various variable values

```
1 $a = 10;
2 $msg = "a has the value $a\n";
3 print $msg; //prints a has the value 10
4
5 $a = 10;
6 $b = "10";
7
8 if($a == $b) { ... } //true
9 if($a != $b) { ... } //false
10
11 if($a === $b) { ... } //false
12 if($a !== $b) { ... } //true
13
14 $a = 1.0; //a is a double
15 $b = 1;   //b is an integer
16
17 if($a == $b) { ... } //true
18 if($a !== $b) { ... } //false
```

4 Strings

- Can use == or ===
• *Should* use comparison functions: `strcmp($a, $b)`, `strcasecmp($a, $b)` (case insensitive)

• Return something negative, zero, something positive depending on relative ordering of `$a`, `$b`

• String variables can be treated as arrays to access individual characters: `$str[0] = 'H';`

• Selected functions:
  - `strlen($str)` – length of `$str`
  - `explode($delimiter, $str, [$limit])` – returns a tokenized array of strings along the provided delimiter (*split* is similar but deprecated)
  - `preg_split($pattern, $str, [$limit])` – splits along a regular expression
  - `substr($str, $start_index, [$length])` – returns the substring starting at `$start_index` with an optional length parameter
  - `str_replace($needle, $replace, $haystack)` – replaces instances of the needle with the replace value (haystack is the input string)
  - `preg_replace($pattern, $replacement, $haystack)` – same as above, but uses regular expressions

```php
$msg = "hello World how are you?";
$msg[0] = 'H';
print "msg has " . strlen($msg) . " characters\n"; //24 characters

$data = "123, john, smith, 456";
$tokens = explode("", $data); //{"123", "john", "smith", "456"}

//split along all whitespace:
$tokens = preg_split("\s", $msg); //{"Hello", "World", "how", "are", "you?"}

$str = substr($msg, 6, 5); //"World"

$str = str_replace("", ":", $data); //"123;john;smith;456"

$str = preg_replace("[\s]", ":", $msg); //"Hello_World_how_are_you?"
```
5 Arrays

- Arrays in PHP are actually *associative arrays*: unordered maps
  - All elements are key-value pairs
  - Keys may be integers or strings
  - Not necessarily zero-indexed nor contiguous
  - Values can be any mixed type

- Initialization syntax

- Using strings as keys: values are type juggled into integers (doubles are truncated to integers)

- Removing items: use `unset()`

- Adding items: you can provide a key (index) or not (assigns a key value equal to the current maximum integer key plus one)

- Accessing: index using a key; missing items result in warnings and a result of `null`

- Selected functions:
  - `print_r($arr)` – pretty prints the contents of an array
  - `array_keys($arr)` – returns a zero-indexed containing keys in `$arr`
  - `array_values($arr)` – returns a zero-indexed array containing values in `$arr`
  - `count($arr)` – returns the number of elements contained in the array

- Since elements may not be contiguous nor even integer keys, we need another way to iterate over elements: `foreach` loops

- Operations:
  - `==` – equality (same key-value pairs)
  - `===` – strict equality (same key-value pairs, same ordering and same types)

- Multidimensional arrays are supported, `$matrix[0][2]`
$myArr = array(5, "bar", 3.14, "ten");

$arr = array(
    "foo" => 5,
    4 => "bar",
    0 => 3.14,
    "baz" => "ten"
);

//delete one element:
unset($arr[4]);

//delete the entire array:
unset($arr);

$arr['five'] = 5;
$arr[] = 12;

//for each key value pair:
foreach($arr as $key => $val) {
    print "$key maps to $val 
";
}

//for each key:
foreach(array_keys($arr) as $key) {
    print "$key maps to " . $arr[$key] . " 
";
}

//for each value:
foreach($arr as $val) {
    print "$val 
";
}

6 Conditionals

The standard if-else-if structures are supported. Switch statements are supported (with integers or strings).
if($a !== $b) {
    print "unequal!";
}

if(strcmp($lastNameA, $lastNameB) == 0) {
    $result = strcmp($firstNameA, $firstNameB);
} else {
    $result = strcmp($lastNameA, $lastNameB);
}

if($a < $b) {
    print "a is less\n";
} else if($a == $b) {
    print "they are equal\n";
} else {
    print "b is less\n"
}

7  Loops

In addition to the `foreach` loops, the standard for, while, and do-while loops are supported

for($i=0; $i<10; $i++) {
    print $i . "\n";
}

$i = 0;
while($i < 10) {
    print $i . "\n";
    $i++;
}

$i = 0;
do {
    print $i . "\n";
    $i++;
} while($i < 10);
8 Input/Output

8.1 Command Line Arguments

Most programs do not involve a user interface or even involve human interaction. Instead, programs executed from the command line can be configured by specifying command line arguments which are passed to the program when invoking it. For example:

```
~> ./programName argument1 argument2
```

When a PHP script is executed through the PHP interpreter, arguments can be passed to the script in a similar manner:

```
~> php PHPScript.php arg1 arg2 arg3
```

Note:

- Arguments are accessible through the `$_argv` array
- For PHP the first argument, `$_argv[0]` is the script name
- The number of arguments can be determined using the count function:

```
count($_argv)
```

```php
<?php
/**
 * This program demonstrates command line argument usage
 * by printing out all the arguments passed to it when invoked.
 */

$n = count($_argv);
for($i=0; $i<$n; $i++) {
    print "argv[" . $i . "] = " . $_argv[$i] . "\n";
}
?>
```

8.2 Standard I/O

- Standard input: `fgets(STDIN,[bytes_length])`: reads a line from the standard input
- Standard output: `print, echo, printf`
- File input:
Open a file handle: `$_h = fopen("filename", "r")` for reading

Read a line (retains endline character): `$_line = fgets($_h)`

End of file testing: `feof($_h)`

Close: `fclose($_h)`

Cool: you can open URLs! `fopen("http://cse.unl.edu", "r")`

• File output:

  Open a file for writing: `fopen("filename", "w")`

  Write a string: `fputs($_handle, $_str)` (or fwrite)

  Close (can also write to a URL to do a post!)

```php
for($_i=0; $_i<count($_argv); $_i++) {
    printf("argv[%d] = %s\n", $_i, $_argv[$_i]);
}

print "Please enter n: ";
$_input = fgets(STDIN);
$_n = intval($_intval);

$_h = fopen("input.data", "r");
while(!feof($_h)) {
    $_line = fgets($_h);
    print $_line;
}
fclose($_h);
```

9 Functions

• Defining (no prototypes):
  ```
  function name(parameters) { ... }
  ```

• No overloading: each identifier can only be used once (redefining is a fatal error)

• Passing by value: default

• Passing by reference: use `&` on parameters
  ```
  function byRef(&$_a, &$_b) { ... }
  ```

• Calling functions: passing parameters is `optional`
• If a parameter is not used, it takes on a null value in the function

• You can define default values other than null:
  function func($a = default_value) { ... }

• No return type is specified, done via explicit return (null implicitly returned)

• Scope
  – By default, all variables are local; need to declare them as global if you want the global version: global $a;
  – Alternatives: $GLOBALS[], array, superglobals $_POST[], etc.
  – Globals are evil.

• Some support for functions as first-class citizens

```
function swap(&$a, &$b) {
    $tmp = $a;
    $a = $b;
    $b = $tmp;
}

function foo($a, $b = 10, $c = 20) {
    return $a + $b + $c;
}

$x = foo(1, 2, 3);
print $x . "\n"; //6

$x = foo(1, 2);
print $x . "\n"; //23

$x = foo(1);
print $x . "\n"; //31

$x = foo();
print $x . "\n"; //30, a is treated as null, casted to zero

//variables can be "set" to functions:
function bar($a) { print $a; }
$y = "bar";
$y(10);
```
10 Searching & Sorting

- Basic sort: `sort($arr)`
- Sorting with a callback: `usort($arr, callback)`
- Searching: `array_search($needle, $haystack)` returns the (first) key corresponding to `$needle`

```php
function myCmp($a, $b) {
    return $a - $b;
}
usort($arr, "myCmp");
```

11 Exceptions

- Built-in class: `Exception`
- Throwing: `throw new Exception("exception message");`
- Handling: try-catch blocks (finally blocks added as of version 5.5)

```php
if($x == 0) {
    throw new Exception("cannot divide by zero");
} else {
    $a = $b / $x;
}

try {
    //potentially dangerous code here
} catch (Exception $e) {
    //handle it here
    print "Exception occurred: " . $e->getMessage();
    exit(1);
}
```
12 Classes

- Declaration
- Instantiation (**new** operator)
- Constructor (only one is allowed)
- Parent constructors are not called implicitly, must use **parent::$_construct()**
- Destructors (supported, **destruct()**)
- Other “magic methods”: **__toString()**
- Member variables and methods
- Visibility keywords: **public, private, protected**
- Static variables and methods: **static**
- Invoking methods: arrow operator, **->**
- Open recursion: **$this** keyword
- Static open recursion: **ClassName::func()** or **self::func()**
- Subclassing using the **extends** keyword, method overriding
class Student {
    private $nuid;
    private $firstName;
    private $lastName;

    public function __construct($nuid, $fName, $lName) {
        $this->nuid = $nuid;
        $this->firstName = $fName;
        $this->lastName = $lName;
    }

    public getFormattedName() {
        return $this->lastName . "", " . $this->firstName;
    }

    public __toString() {
        return $this->getFormattedName . " (" . $this->nuid . ")";
    }

    // instantiate:
    $s = new Student(1234, "John", "Smith");
    print "Student: " . $s->getFormattedName() . "\n";
}