Hypertext PreProcessor (PHP)

PIC 40A, UCLA

PHP

PHP, an acronym for **Hypertext PreProcessor**, is a popular programming language for the server-side of a website (roughly **80%** of websites use PHP for the client-end).

There was a time long ago when the acronym made more sense, standing for **Personal HomePage**.

For the most part, it is a series of scripts that generate HTML that a web browser displays, without the clients ever seeing the code that goes on behind the scenes.

Like JavaScript, it is a high level language. It is also an interpreted language: what you write as a programmer is parsed by a runtime engine.

PHP Setup

PHP code is nested between **<?php** and **?>**, which is sometimes abbreviated as **<?** and **?>.** The first is more robust, however, so should be used.

For PHP to run on a webpage, the very first line should be a link to the binary that compiles the program! For many servers (and PIC) it is:

#!/usr/local/bin/php

If that is not there at the very top of the page, it will not run PHP!

PHP Setup

Many servers are based with a Linux operating system and the PHP engine may stumble and generate horrific errors if it runs across Windows-based return carriages, etc.

If a PHP file was written in a Windows editor, after being transferred to the server, it should be run through the Linux command:

```
dos2unix file_name.php
```

Or all files can be run like this together with:

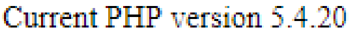
```
dos2unix *
```

In addition, for PHP to run, they must have properties **755**. When the server runs PHP scripts, it runs as **wwwrun** or some other "outside" user, and without proper permissions, the scripts can’t run.

Simple PHP Program

Here is a simple PHP program to display the version of PHP run by the engine:

```
#!/usr/local/bin/php
<!DOCTYPE html>
<html>
<head>
  <title>PHP Demo</title>
</head>
<body>
  <main>
    <?php
      # display our version of PHP
      echo '<p>Current PHP version ', phpversion(), '</p>';
    ?>
  </main>
</body>
</html>
```



```
Current PHP version 5.4.20
```

Simple PHP Program

This PHP file looks a lot like a standard HTML file. At the top we link to the PHP binary. We write normal HTML code.

Interspersed in the code is PHP code, enclosed in the **<?php ... ?>** which produces more HTML that the browser has to parse.

Simple PHP Program

In PHP single line comments can be prefixed by either **#** (as in the example) or //.

Multiline comments begin with /* and end with */.

The **phpversion** function outputs the current version of PHP being run.

Simple PHP Program

echo is not a function, but a language construct. It will print everything until there is a semicolon. It can take multiple inputs by separating them with a comma.

Closely related is **print**. It always returns the value 1. **print** can only take one argument.

Both **echo** and **print** can be called with or without parentheses. **echo** is marginally faster than print.

Neither are real functions, which is why they can be called without parentheses.

Simple PHP Program

From a visitor’s perspective, they really just see HTML. Here’s what happens if we view the page source...

```
<!DOCTYPE html>
<html>
<head>
  <title>PHP Demo</title>
</head>
<body>
  <main>
    Current PHP version 5.5.14
  </main>
</body>
</html>
```

Data Types

In PHP there are four scalar types:

- **boolean**
- **integer**
- **float**
- **string**

PHP also has

- **arrays**
- **objects**
- **NULL**
- **resources**

Variable Names

In PHP, **all variables begin with a $.** They can then contain any number of underscores, letters, or digits.

```
$x = 42; // x is a variable storing 42
```

Warning: if the dollar sign is forgotten, all sorts of bugs can come up where things aren't printed, values aren't used, etc. All variables must have the dollar sign.

Data Types: Boolean, Integer, Float

In PHP, the boolean values are **true** and **false.** These reserved keywords are case insensitive: **True** and **False** also work, etc.

If a number is given without a decimal in it, it is taken to be an integer. With a decimal, it is a floating value.

```
$x = true; // boolean
$y = 11; // integer
$z = 0.29; // float
```

Data Types: Boolean, Integer, Float

All of the standard C++/JS operators exist in PHP: **+**, **-**, *, /, **%**, **++**, **--**, **+=**, **-=**, ***=**, /**=**, and **%=.**

For **+**, **-**, and * between two integers, the result is always an integer. For /, if the result would involve a decimal, it will be a float.

The mod operators **%** and **%=** coerce their arguments to integers if not already integers.

Math

PHP has some functions that are defined in a global namespace, including:

- **abs** (absolute value)
- **sin, cos, tan** (sine, cosine, tangent)
- **exp, log** (natural exponential, natural logarithm)
- **pow** (raises first argument to power of second argument)
- **round** (rounds to nearest integer)
- **ceil, floor** (integer ceiling and floor functions)

Math

Random number in PHP require multiple stages. First, we must **seed** the random function:

```
mt_srand();
```

The random numbers will be integers that are uniformly distributed from 0 to some maximum value. That maximum value is returned from **mt_getrandmax**:

```
$max_rand = mt_getrandmax();
```

To get a random integer in the range, we use **mt_rand**:

```
$val = mt_rand() / ($max_rand+1); // will be from 0 to 1

$die_roll = floor(6*$val) + 1;
```

Data Types: String

Strings are usually (some bizarre exceptions exist) enclosed in either single or double quotes. The two are not always the same. For plain text, the single quote is preferred because content within double quotes goes through a more thorough parsing.

The **.** operator concatenates strings.

```
$x = ’hello, ’;
$y = ’world!’;

echo $x, $y; // prints "hello, world!"
```

Data Types: String

Escape sequences are permitted within strings.

For a single quoted string, the only permitted escape sequences are \’ and \\ for a ’ and \, respectively. All other backslahes are interpreted literally.

Double quotes allow for more escape sequences like **’** for ", **t** for a tab, **n** for a new line. But they also allow variables to be evaluated within the string.

```
$year = 2019;
echo "The year is $year."; // prints "The year is 2019."
echo ’The year is $year.’; // prints "The year is $year."
```

To print $ within double quotes, we use **$**.

Data Types: String

Strings are indexed from 0.

```
$msg = ’hello world’;
echo $msg[4]; // prints "o"
```

There are a number of handy functions to use for strings:

```
echo substr($msg, 6); // prints "world";
```

The **substr** takes a string argument, a starting index, and an optional length. Its behaviour is similar to that of C++ and JavaScript, but we are not calling a member function.

Data Types: String

```
$msg = ’hello world’;

strlen($msg); // 11
strpos($msg, ’world’); // 6
```

strlen gives the length of a string.

strpos searches a string from the beginning for a piece of text returning the index where it begins. If the string is not found, it returns **FALSE**.

Remark: obviously this can be dangerous. Indeed **FALSE** can very easily be converted to **0**. As with JavaScript, we will need === and !== operators.

Data Types: String

String concatenation is done via the **.** and **.=** operators with the latter including assignment.

```
$x = 'ab';
$y = 'c';
$z = $x . $y; // $z === 'abc'

$msg = 'hello';
$msg .= ' world'; // now $msg === 'hello world'
```

Remark: yes, there are === and related comparisons in PHP...

Data Types: String

```
$msg = ’hello world’;

$arr = explode(’ ’, $msg); // array, [’hello’, ’world’];
$msg2 = implode(’ ’, $arr); // string ’hello world’
```

The **explode** function takes a delimiter and a string, splitting that string into parts of an array.

The **implode** function takes a delimiter and an array, making a string out of it, with that delimiter placed between elements.

Data Types: Constants

PHP allows for scalar types to be kept constant. We use the **define** syntax:

```
define('FOO',19);
```

Above, we defined the symbol **FOO** to be the value 19.

Constants are often capitalized by convention. Valid names begin with a letter or underscore, and then contain any number of letters, digits, or underscores.

Remark: in newer versions of PHP, it is also possible to write

```
const FOO = 19;
```

In C++, this is just like specifying a preprocessor directive with define:

```
#define FOO 19
```

PHP Comparisons

Logical negation is done with **!**.

We use **<** for less than, **<=** for less than or equal, **>** for greater than, **>=** for greater than or equal.

With **==**, two values are **equal**, after possible coercion, and both **!=** and **<>** mean two values are not equal, even after coercion.

With **===**, two values are **identical**, i.e., equal and of the same type and **!==** signifies either the types differ or the two values are not equal.

This **identical** can go a bit too bar. In PHP, **0 === 0.0** is **false** because we are comparing an integer and a floating point!

PHP Coercions

PHP uses the same sorts of coercions as JS for the most part in determining whether something is **truthy** or **falsey** and when comparing between different types it can try to convert them to numbers. But with PHP, "0" is **falsey** and an empty array is also considered **falsey.**

Some other fascinatingly annoying coercions: the booleans **true** and **false** can be coerced into the strings ‘1’ and ‘’ (empty string!). Thus if we ever **print** or **echo false**, nothing gets printed...

In PHP these coercions are often called **type juggling.**

PHP Coercions

When trying to convert a string to a number, PHP will extract the numeric part of the string starting from the beginning, ignoring initial white space, stopping when it reaches letters that don’t make sense. But it will read scientific notation, too. And the empty string is converted to 0.

In PHP, **"php" == 0** is **true**!!!

PHP processes **+**, **-**, *****, /, **%**, and **.** from left to right. With **.** it will convert the operands to strings; otherwise it will convert the operands to numbers.

PHP Coercions

```
3 + ’ 5hello’ . 7; // ’87’: (3+5) . 7 => 8 . 7 => ’87’
12 . 3 - ’4’; // 119: (12 . 3) - ’4’ => ’123’ - ’4’ => 123-4 = 119

’hello6’ + 7; // 7: 0 + 7
’4hello’ * 7; // 28: 4 * 7
’1.2e+2hello’ + 7; // 127: 120 + 7
’hello’ . 7; // ’hello7’: concatenation
4 . 7; // ’47’: concatenation
```

PHP Coercions

To view the type of a variable, we use **gettype**:

```
$x = true;
$y = ’red’;

gettype($x); // boolean
gettype($y); // string
```

Data Types: Array

Like with JavaScript, PHP arrays can store anything, even functions. We use the **array** function to create an array:

```
$x = array(4,true,’hello’); // stores value 4, true, and ’hello’
```

And, like strings, arrays have 0-based indexing.

```
echo $x[2]: // prints "hello"
```

Data Types: Array

Some useful functions with arrays:

```
$x = array(’c’,’b’,’a’);
$new_length = array_push($x, ’z’, ’y’); // $x = [’c’,’b’,’a’,’z’,’y’];
$last_entry = array_pop($x); // $x = [’c’,’b’,’a’,’z’]
echo count($x); // 4
sort($x); // sorts the array alphabetically (all items are strings)
```

The **array_push** function accepts an array and an arbitrary list of values to add to the end, returning the new length.

The **array_pop** accepts an array and removes its last element, returning that value.

The **count** function accepts an array and returns its number of elements.

The **sort** function accepts an array and sorts it in ascending order.

Data Types: Array

Remark: if **sort** is called on an array with a mix of data types, the sorting is apparently not stable.

```
$x = array(2., 1, false, ’0’, ’-19’);
sort($x); // [’0’, false, ’-19’, 1, 2.]
```

To do more complicated sorting, we need to use the **usort** function...

Data Types: Array

Secretely all PHP arrays are **associative**, i.e., having key-value pairs. By default the keys are just indices. The keys can be either **string**s or **integer**s.

```
$arr = array('foo'=>'bar', 'cat'=>'meow', 4=>7, 9.8=>false);

// so we have
$arr['foo']; // 'bar'
$arr['cat']; // 'meow'
$arr[4]; // 7
$arr[9]; // false;
```

We can also add new elements to this array by subscripting it:

```
$arr['colour'] = 'red';
```

Data Types: Array

A newer piece of syntax, more consistent with JavaScript is creating an array with square brackets like:

```
$x = [2,4,6,true];
```

Data Types: Array

There is also the **[]** syntax to add an element at the end of an array. It is more efficient than calling **array_push** if only a single element is being added.

```
$x = array(1,2,3);
$x[] = 4; // now last element is 4
```

To remove an element from an array, we use **unset**:

```
unset($x[2]); // now $x = [1,2,4], removed item index 2
```

Warning: this can have weird effects upon the array indices.

Remark: a variable can be deleted and removed from the namespace by calling **unset** upon it, but there are certain contexts where **unset** will have no effect.

Data Types: Array

For arrays indexed by integers, the simple remedy to the “missing" indices when an element is **unset** is to use the **array_values** function.

```
$x = ['a', 'b', 'c'];

unset($x[1]); // so now $x is { 0=>'a', 2=> 'c'}

$x = array_values($x); // now $x is { 0=>'a', 1=> 'c'}
```

Data Types: Object

An **object** is the equivalent of the C++ class structure. Objects can store properties (member variables) and methods (member functions).

Given an object handle, properties and methods are accessed via **->** (the equivalent of the **.** access in JavaScript and C++).

```
$x = new Square(3);
echo $x->get_area();
```

We will study how to write them later.

Data Types: NULL

NULL is the special value and the data type **null** that represents the state of not having a value. This is like the JavaScript **null** or **undefined.**

```
$a = NULL; // has no value
```

Data Types: Resource

A **resource** is a variable that references an external resource (file).

File pointer resources (like the C++ **std::ifstream**, **std::ofstream**) allow us to read from and write to files would be one example. A link to a **database** through which we can make queries would be another example.

Type Juggling

In PHP, we can explicitly cast one variable type to another using the C-style cast syntax. The following conversions are supported by PHP: conversion to **bool**, boolean; **int**, integer; **float**, float; **string**, string; and **array**, array.

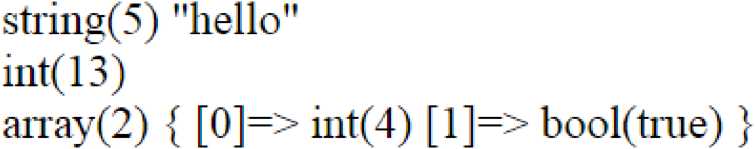
For example:

```
$x = 4.14;
$y = false;

$a = (int) $x; // $a is 4
$b = (string) $y; // $b is ‘’
$c = (bool) "hello"; // $c is true
```

var_dump

For debugging purposes, it can be handy to display the value of a variable. For that, we can use **var_dump**.

```
$x = "hello";
$y = 13;
$z = array(4,true);

var_dump($x);
var_dump($y);
var_dump($z);
```



```
string(5) "hello"
int(13)
array(2) { [0]=> int(4) [1]=> bool(true) }
```

References vs Values

PHP supports both passing by reference and passing by value.

In code, to make one variable a reference to another, we use the **&** operator:

```
$x = 11;
$y = &$x;
++$y; // both $x and $y are 12
```

References vs Values

For a function, the input arguments need to have an & to accept by reference.

```
function does_nothing($x) { ++$x; }

function does_something(&$x) { ++$x; }
```

Function Default Arguments

PHP supports scalar values, arrays, and **NULL** values as default arguments for functions.

```
function foo($x, $y = array(1,2,3), $z = 111){
   echo $x + $z;
}
```

Note: default arguments must be specified from right to left.

```
foo(3); // 114
foo(3,4); // 114
foo(3,4,5); // 8

foo(); // PHP Warning: missing 1 argument for foo...
```

Global Variables in Functions

In PHP, if a variable is defined in a function, it is only known within that scope. Also, only variables passed as parameters are known to the function by default. It will not be able to access a global variable unless we request so.

```
$some_global = 111;

function foo($y) {
   global $some_global; // requesting access to the global
   $z = $some_global + $y; // this is okay
   echo $z;
}

echo $z; // ERROR: z not defined here!
```

Global Variables in Functions

Alternatively, we can access the value from the **$GLOBALS** variable:

```
$some_global = 111;
function bar($y) {
   echo $GLOBALS[’some_global’] + $y;
   $GLOBALS[’z’] = $GLOBALS[’some_global’] + $y; // adds to globals
}
```

The **$GLOBALS** variable stores all global variables. Above, we could access **$some_global** by the index **’some_global’** without the **$.** We also added **$z** as a global variable so that the code below is okay:

```
echo $z;
```

Anonymous Functions

In PHP, there are also function objects that can be created on the fly. Here are a few examples too see the syntax.

```php
$fun = function($array, $name){ // anonymous function stored as $fun
  $array[0]();
  $array[1]($name);
};

$arr = array( // this array stores functions
  function() { echo 'hello '; },
  function($name) { echo $name; }
  );

$fun($arr, "Joe"); // prints "hello Joe"
```

Anonymous Functions

Just as in JavaScript and C++, we can use anonymous functions as call parameters to sort lists.

```php
$names = array('Colleen', 'Bryce', 'Adam', 'Dan', 'Ella');

// we will sort by name length!
usort($names,
  function($x,$y)
    { return strlen($x) - strlen($y);}
  );
```

The **usort** function takes an array to sort as its first argument and a function as its second argument. The comparison function *must* return a negative value, zero value, positive value, respectively, depending on whether its first argument is less than, "equal to", or greater than the second argument, respectively, based on our sorting criteria.

Anonymous Functions

Alternatively, we could have written a free function and passed it as a parameter in quotes.

```php
function sort_lengths($x,$y){
  return strlen($x) - strlen($y);
}

$names = array("Colleen", "Bryce", "Adam", "Dan", "Ella");

// we will sort by name length!
usort($names, "sort_lengths");
```

Function Documentation

We will focus on a simple documentation style similar to C++ and JS.

```
/**
description

@param type $name description
...
@return description
*/
```

```
/**
This function compares the lengths of two strings

@param string $x the first string
@param string $y the second string

@return the length of $x minus the length of $y
*/
function sort_lengths($x,$y){
  return strlen($x) - strlen($y);
}
```

Functions Accepting or Returning References

PHP adheres to a **write on copy** optimization. This means that passing by reference is only required if the intent is to modify the argument! It is not done for efficiency.

PHP will pass arguments by reference and only if they are modified within the function does that turn into a pass by value. **But objects acquired through “new” behave like pointers so there are subtleties**. More to come in the object oriented discussion.

In C++, we very often pass an object to a function as a reference to const, even if it is not being modified to avoid an unnecessary copy of an object.

```
bool checks_string(const std::string& input);
```

Function Arguments

Like JavaScript, PHP can accept arbitrary numbers of arguments. Within a function body, **func_num_args()** returns the number of arguments that were passed. And **func_get_args()** returns an array of the values passed.

```
function foo() {
   $len = func_num_args();
   $inputs = func_get_args();

   for($i=0; $i<$len; ++$i){
      var_dump($inputs[$i]);
   }
}
```

Function Ordering

Like JavaScript, a function need not be declared/defined before it is referenced and used. But it should be defined somewhere.

```
// works fine:

function foo() { bar(); }

foo();

function bar() { echo ’11’; }
```

Closures

PHP allows for **closures** but we need to request the closure by specifying the capture arguments with **use (list of arguments).**

```
function transform($m,$b) {
   return function($x) use ($m,$b) { return $m * $x + $b; }
}

$celsius_to_fahrenheit = transform(1.8, 32);
$winnipeg_mild_winter_in_F = $celsius_to_fahrenheit(-25); // -13
```

With C++ the "use" appears as the capture list in square brackets:

```
auto transform(double m, double b){
  return [m,b](double x)->double{ return m*x+b; };
}
```

Control Flow: if/else I

All programming languages need to have control flow and here we'll look at the control flow structore of PHP. We start with **if** statements:

```
if(condition) {
   // do stuff
}

if(condition) {
   // do this
}
else{
   // or this
}
```

Control Flow: if/else II

```
if(condition) {
   // do this
}
elseif{
   // this
}
else {
   // or this
}
```

In PHP, **elseif** is a thing, meaning the same thing as **else if** but it applies more widely than **else if.**

Control Flow: if/else

Warning: using correct variable names is really important in PHP. Forgetting a dollar sign can be an absolute nightmare!

```
$reveal_deep_dark_secret = false; // don’t share it!!!

if( reveal_deep_dark_secret ) {
   echo ’revealing secret now...’;
}
```

Above, the secret gets revealed. The missing $ in front of **$reveal_deep_dark_secret** in the **if** condition ends up treating the condition as the string ’reveal_deep_dark_secret’ which is coerced to **true** because it is nonempty and not ’0’...

Control Flow: if/else

More warnings: Here's another thing to be wary of... let's suppose you did remember the $ but made a typo somewhere else...

```
$did_good_job = true;

if($did_goodjob){ // oops, typo..., definitely not true now...
   echo 'well done';
} else{
   echo 'hang your head in shame!';
}
```

Something PHP knows is a variable but which has not been set yet is treated as **NULL** (which is coerced to **false**).

Control Flow: for/while/do

There are really no surprises for these loops.

```
for(initializations; condition; after body steps) {
   // do stuff
}

while(condition) {
   // do stuff
}

do {
   // stuff
} while(condition);
```

Control Flow: foreach

Like the C++ range-for and JS **for of**, PHP has **foreach** to iterate through arrays and objects.

```
$arr = [’a’=>1, ’b’=>2, ’c’=>3];

// with this syntax, we go through the values 1, 2, 3 calling them $val
foreach($arr as $val) { // prints: "1 2 3"
    echo $val, ’ ’;
}

// with this syntax, we track the keys and values
// for each iteration, they key is $key and the value is $val
foreach($arr as $key=>$val) { // prints: "a=>1 b=>2 c=>3"
    echo "$key=>$val ";
}
```

Note: we used double quotes above to evaluate the variables.

Control Flow: foreach

Remark: there is nothing special about the names **$key** and **$val**: they could be named anything. They are names we use in the loop.

Unfortunately in PHP, variables defined within a loop are also defined after a look executes...

Control Flow: foreach

```
// with this syntax, we change the array’s values
foreach($arr as &$val) { // prints: "1 2 3"
   $val *= 2;
}

unset($val); // break $val from referencing last element
```

Using an **&** allows us to go through the elements by reference but we must remember to **unset** the loop variable. Generally variables defined in loops will live beyond the loop... Above, **$val** no longer exists.

As in C++ and JS, **if**-statements, **for**, **while**, and **foreach** loops do not require braces to be grammatically correct if the body is a single statement. But for robustness, they should be included! Also each control flow branch should be commented.

Control Flow with Colons

PHP has an alternative syntax for control flow. Instead of, for example:

```
if($x) {
   ++$y;
   echo $y;
} else {
   --$y;
}
```

we could write:

```
if($x) :
   ++$y;
   echo $y;
else:
   --$y;
endif;
```

Control Flow with Colons

With this, braces are not required, even for multiple statements in a control flow branch. Each opening brace is replaced by **:**.

But without braces, it's impossible to know when a block ends, so the parser searches for a follow-up control flow keyword like **else**... and at the end of the day, something has to signify the end of the control flow structure so there is, as in this example, and **endif**.

The requisite endings are **endif;**, **endwhile;**, and **endforeach;**.

And when using the colon syntax, **else if** is not allowed but **elseif** is.

Control Flow with Colons

Danger: while **elseif** and **else if** work fine for the normal braced control flow structures, **else if** is not valid for control flow structures that use the colon.

Danger: do not mixed the syntaxes of the colon or non-colon control flow. It is an error to mix the two syntaxes in a single control flow statement; it is also likely to throw an error to mix the syntaxes between inner and outer brances of control flow structures.

Control Flow: Interspersing HTML and PHP

We can intersperse HTML and PHP. Below, **"ZYX"** gets printed. Then 10 lines of **"hello"**.

```
#!/usr/local/bin/php
<?php
$str = "0";
if($str){ // recall that the string "0" is falsey in PHP ?>
  <p>ABC</p>
  <?php
} else { // so execute this ?>
  <p>ZYX</p>
  <?php
}

for($i=0; $i<10; ++$i) { // do this 10 times ?>
  <p>Hello</p> <?php
}
?>
```

Effectively HTML without being contained inside of PHP tags is treated like an **echo** and just gets rendered directly.

HTTP Requests

HyperText Transfer Protocol (HTTP) requests allow for clients to request information/data from a server.

With a **get** method, a query string is generated and appears as part of the URL the user sees. It is quite common, but is not secure and sensitive data like passwords, etc., should never be used with it. A **get** method can only be used to request data, never to modify it.

With a **post** method, a query is sent, but it is "packed in a sealed envelope", as a kind of analogy. No one can see the request. These methods can create/modify data on the server.

There are others like **put**, **delete**, etc., but we won’t get into them.

Superglobals - $_SERVER

We have already encountered **$GLOBALS**, one of the superglobals of PHP. In managing HTTP requests, a few more become important.

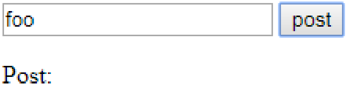
$_SERVER stores information about file names, script locations, etc. Two of its most useful members are:
$_SERVER['PHP_SELF'], the current php page/script, and
$_SERVER['REQUEST_METHOD'], the method of access: GET, POST, etc. For example,

```
if ($_SERVER['REQUEST_METHOD'] === "POST") { /* do stuff */ }
```

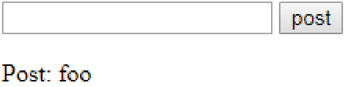
Superglobals - $_POST

$_POST stores data from form when **method=post** is used. We access form data by indexing this variable for the names of the form elements!

```
<form method="post" action="<?php echo $_SERVER['PHP_SELF']; ?>" >
  <input type="text" name="userMessagePost" />
  <input type="submit" value = "post" />
</form>

<p>
<?php
  echo "Post: ", $_POST['userMessagePost'], "<br/>";
?>
</p>
```



Post:



Post: foo

Superglobals - $_POST

A remark is worth remarking on the **action=** part of the web form.

In a web form, an attribute **action** can be specified. This action can be a few lines of JavaScript or a link to a script to run, PHP or JavaScript.

If we wanted to run a PHP script on another page, we could write **action="that_other_page.php"** and redirect the user and their information over there.

To allow the PHP script to run on the current page, we use **$_SERVER['PHP_SELF']** to reference the current PHP script. And by **echo**ing that in the php script in the quotes, we refer the action to the same page.

Superglobals - $_GET

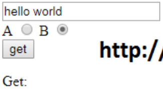
$_GET stores data from form when **method=get** is used. We access form data by indexing this variable for the names of the form elements!

With **get**, we see the name-value pairs appearing in the URL in the form:

somephpscript.php?var1=value1&var2=value2

```
<form method="get" action="<?php echo $_SERVER['PHP_SELF']; ?>" >
  <input type="text" name="userMessageGet" />
  <br/>
  <label for="A">A</label> <input type="radio" name="letters" value="A" id="A" />
  <label for="B">B</label> <input type="radio" name="letters" value="B" id="B" />
  <br/>
  <input type="submit" value = "get" />
</form>

<p>
<?php
  echo "Get: ";
  if(isset($_GET['userMessageGet'])){ // check if set
    echo $_GET['userMessageGet'], "<br/>";
  }
  if(isset($_GET['letters'])){
     $_GET['letters'], "<br/>";
  }
?>
</p>
```

Superglobals - $_GET

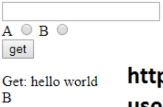


hello world

A ○ B ●

Get:

http://www.page.com/folder/index.php



A ○ B ○

Get: hello world
B

http://www.page.com/folder/index.php?
userMessageGet=hello+world&letters=B

Superglobals - $_GET

Remark: simply by appending a query string **?name=value**, etc., to the end of a PHP script name, the **$_GET** superglobal is populated with the property names and values.

www.someplace.com/somepage.php?name=foo

```
#!/usr/local/bin/php
<!DOCTYPE html>
<html>
<head>
  <!-- page title will be foo -->
  <title><?php echo $_GET['name']; ?></title>
</head>
</html>
```

Arrays in Forms I

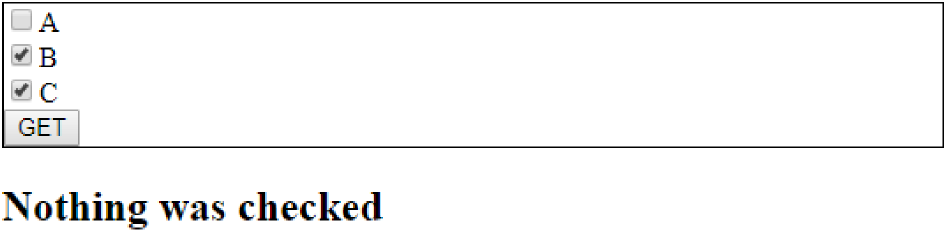
Recall that for web forms that have checkboxes or select fields where multiple values can be submitted, it is necessary to give a **name="something[]"** for that collection of options. Using the array syntax ensures that an array is captured. The array could be empty if no options are selected. If multiple options are selected, there will be an array carrying all the values that were selected.

```
<form id="byGet" method="get" action="<?php echo $_SERVER['PHP_SELF']; ?>">
  <input type="checkbox" name="x[]" value="A">A<br/>
  <input type="checkbox" name="x[]" value="B">B<br/>
  <input type="checkbox" name="x[]" value="C">C<br/>
<input type="submit" value="GET" />
</form>

<p>
<?php
  if(isset($_GET['x'])){ // ensure was set
    $checks = $_GET['x']; // store array as $checks
    if (count($checks)===0){ // size 0 means no checks
      ?>
      <h2>Nothing was checked</h2>
    <?php
    }
    else{ // something was selected
      foreach($checks as $selected){ // go through each selection
        echo $selected, " was chosen. ";
      }
```

Arrays in Forms II

```
        }
    }
?>
</p>
```

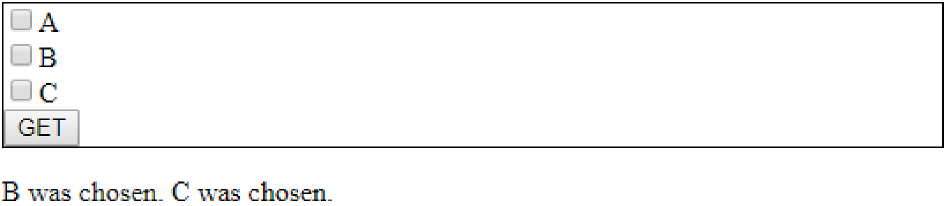
Arrays in Forms



☐ A
☑ B
☑ C

GET

Nothing was checked



☐ A
☐ B
☐ C

GET

B was chosen. C was chosen.

Superglobals - $_COOKIE I

In PHP, there is a global object **$_COOKIE** that has access to cookie data. Unfortunately it is even harder to use than cookies are to use on the JS-side. Here is a simple example.

page.php

```
#!/usr/local/bin/php
<?php
  define('DAY', 86400); # seconds per day
  $name = 'user';
  $value = 'Joe Bruin';
  setcookie($name, $value, time() + DAY);
?>
<!DOCTYPE html>
<html>
<head>
  <title>Page</title>
  <script src="cookie.js" defer></script>
</head>
<body>
  <p>
    <input type="button" value="add cookie" onclick="add();" />
  </p>
  <p>
    <?php
      if(isset($_COOKIE['user']) { // if cookie user set
        echo $_COOKIE['user'], "<br/>";
      }
```

Superglobals - $_COOKIE II

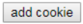
```
      if(isset($_COOKIE['pet']) { // if cookie pet set
        echo $_COOKIE['pet'], "<br/>";
      }
    ?>
  </p>
</body>
</html>
```

cookie.js

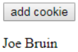
```
function add(){
  let name = "pet";
  let type = "cat";
  let expiry = new Date();
  const day = 24*3600*1000;
  expiry.setTime(expiry.getTime() + day);
  document.cookie = name + "=" + type + "; expires=" + expiry.toUTCString() + "; path='/'";
}
```

Superglobals - $_COOKIE

First time loading the page:

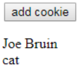


After pressing "show cookie", we see the cookie set by PHP:



Joe Bruin

After pressing "add cookie" then "show cookie", now the JS cookie is also added



Joe Bruin
cat

Superglobals - $_COOKIE

There is a lot going on in this example. It illustrates many subtleties to working with cookies and the differences between PHP and JS handling of cookies.

For PHP, any calls to **setcookie**, a function that sets cookies, must be handled "before any headers are sent". Headers are sent when HTML content, including whitespace, is rendered. The headers are part of the HTTP protocol specifying important information about what information the browser is receiving, etc. Cookie setting *must* come even before **<!DOCTYPE html>**! Failing to do so will cause the page to crash.

Superglobals - $_COOKIE

In PHP the **time** function returns the number of seconds since January 1, 1970, 00:00 GMT time. The **setcookie** accepts arguments of name, value, an expiry time counting seconds since **time 0**, and a location where the cookie is valid. If no location is given, the default location is the current folder. If no time is given, the cookie expires when the browser closes.

Warning: the **isset** function can take anything as an argument. If its argument is not even a variable that has been defined, that's okay, it will just return **false** rather than generate an error. Even something that seemingly should generate an error **isset($COOKIE['pet'])**, mispelling the variable and subscripting a nonexistent variable, will not be an error.

Superglobals - $_COOKIE

To remove a property from a cookie, we can:

```
if( isset($_COOKIE[‘property‘])) { // only remove if actually set!

   unset($_COOKIE[‘property’]); // changes PHP superglobal

   // change brower cookie to expired!
   // be sure to have the same path as original cookie
   setcookie(‘property’, ‘ ’, time() - 3600, ‘/’);
}
```

Reading and Writing Files

The **fopen** function gives us a file pointer resource, allowing us to open a file, either to read from it or to write to it. The flag **"r"** indicates reading, **"w"** indicates writing from the beginning and emptying the file, and **"a"** indicates appending to the end.

If the file does not yet exist, both **"w"** and **"a"** will attempt to create the file.

```
$file = fopen(‘file_name.txt’, ‘w’); // example of opening to overwrite.
```

Reading and Writing Files

If **fopen** fails, it returns the value **false** rather than a file pointer resource. Thus, it is common to see code such as:

```
$file = fopen(‘foo.txt’, ‘r’) or die(‘could not open file’);
```

The **or** keyword pertains to logical or, **||**. If the opening fails then the first expression is false and the second one is evaluated. The **die** function causes the script to **exit** (a termination function) and displays an error message.

Reading and Writing Files

Remark 1: a resource evaluates to **true** and lazy logical evaluation is applied.

Remark 2: in PHP, **or** has lower precedence than **=** (which is lower than ||). So the expression should be seen as:

- Evaluate **fopen(‘foo.txt’, ‘r’)** and assign it to **$file**.
- If that result is a file handling object, which is **truthy**, the lazy evaluation of **or** makes the **or** evaluate to **true** without further processing.
- And if **$file** is **false** then the second term of the **or** is evaluated and the program ends.

Reading and Writing Files

The **fwrite** function takes a file pointer resource and content as arguments and writes them to the file.

```
$file = fopen(’poem.txt’, ’w’) or die(’cannot open’);
fwrite($file, "It is morning now.\nWeb programming is good fun!\nThis
was a haiku");
```

We generate the file **poem.txt**:

```
It is morning now.
Web programming is good fun!
This was a haiku.
```

Remark: the double quotes are important here! With single quotes, line characters would be directly rendered as **n**.

Reading and Writing Files

After using a file pointer resource, it should be closed to avoid having unused resources hanging about. We use **fclose** for that.

```
fclose($file);
```

Reading and Writing Files

The function **feof** returns a boolean value to indicate if the end of file has been reached.

The **fgets** function reads and extracts one line of a file at a time (up to a new line character).

```
$display_poem = fopen('poem.txt', 'r') or die('cannot open file');

while(!feof($display_poem)) { // while still more to read
   $line = fgets($display_poem);
   echo $line, '<br/>';
}
```

Flat File Management

A **flat file** as opposed to something like a **database** is one that stores data in plain, unformatted text, with nothing but forms of whitespace to designate separate pieces of data.

Working with data this way is often a lot simpler but it does come with an overhead of deciding on a format.

As a simple guideline, each record should occupy a single "line" – something that terminates in a new line character. And between field values, tabs can separate data.

The above is very clear: each entire line is one record. And a tab always means we are moving onto another field.

Flat File Management

When we read data from a flat file, we can easily read one record at a time with the **fgets.** But if the fields are separated by tabs, we should also remember the **explode** function to break the string apart**.**

```
$line = fgets($file);

$fields = explode("\t", $line);
```

Superglobals - $_SESSION

Instead of using **cookies** on the user's end, we can store information on the **server** end with **PHP sessions**.

This entails preceding all HTML with a **session_start();** command. Like cookie setting, session setting must occur before headers are sent.

We can add values to the global **$_SESSION** variable. Other PHP scripts that are accessed from a session page also get access to the session variables if they have a **session_start();** command at the top.

The **session_unset();** command clears session variables and **session_destroy();** command ends a session.

Superglobals - $_SESSION

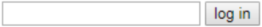
PHP sessions can (and often should) be named to avoid clashes with other PHP sessions. To name (or resume) a PHP session, we use:
session_name('whatever');

The **session_name** function must precede **session_start**. Then the session being started is either given the name prescribed or the session with the given name is resumed.

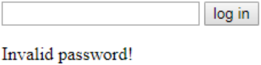
Simple Login System

Here we’ll consider a very simple login system built with PHP. We assume there is a flat file **password.txt** on the server storing the password. Later with hashing and databases this can be made more secure.

When the user first visits **login.php**:

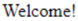


If they enter the wrong password:



Invalid password!

If their password is correct they are taken to **welcome.php**:



Welcome!

Simple Login System I

login.php

```php
#!/usr/local/bin/php
<?php
  session_save_path(dirname(realpath(__FILE__)) . '/sessions/');
  session_name('Demo'); // name the session
  session_start(); // start a session
  $_SESSION['loggedin'] = false; // have not logged in

  /**
  This function validates a password and sets the $_SESSION token to true
  if it is correct, logging them in and sending them to the welcome page.
  Otherwise it flags $error as true.

  @param string $password the password the user entered
  @param boolean $error the error flag to possibly change
  */
  function validate($password, &$error){
    $fin = fopen('password.txt', 'r'); // open file to read
    $true_pass = fgets($fin); // get the line
    fclose($fin); // close the file
    $true_pass = trim($true_pass); // trim white space

    if($password === $true_pass){ // if they match, great
      $_SESSION['loggedin'] = true;
      header('Location: welcome.php');
    }
    else { // bad password
      $error = true;
    }
  }

```

Simple Login System II

```
  $error = false;
  if(isset($_POST['pass'])){ // if something was posted
    validate($_POST['pass'], $error); // check it
  }
  ?>
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Login Page</title>
</head>
<body>
  <main>
    <form method = "post" action ="<?php echo $_SERVER['PHP_SELF']; ?>">
      <input type="password" name="pass" /> <input type="submit" value="log in" />
      <?php if($error) // wrong password { ?>
        <p>Invalid password!</p> <?php
      } ?>
    </form>
  </main>
</body>
</html>
```

Simple Login System III

welcome.php

```
#!/usr/local/bin/php
<?php
  session_save_path(dirname(realpath(__FILE__)) . '/sessions/');
  session_name('Demo'); // resume Demo session
  session_start(); // start a session
?>
<!DOCTYPE html>
<?php
  // either no session or not logged in
  if(!isset($_SESSION['loggedin']) or !$_SESSION['loggedin']) {  ?>
<html>
<head>
  <title>Unwelcome</title>
</head>
<body>
  <p>Go back and log in.</p>
</body>
</html> <?php
  }
  else { // then they are logged in for real ?>
<html>
<head>
  <title>Welcome</title>
</head>
<body>
  <p>Welcome!</p>
</body>
</html> <?php
  } ?>
```

Simple Login System

Some explanation...

Some servers block access to a temporary folder for sessions so we specify a session save path. The **dirname(realpath(__FILE__))** is a special way to obtain the full working directory of the *current PHP* script. In this case, session data will be stored in a **sessions** folder local to where the PHP script runs. We need to create such a folder first and set its permissions to 755.

At the start of **login.php**, we name the session **’Demo’** and start it. We set **$_SESSION[’loggedin’]** to **false** because the visitor is not yet logged in, say. Or we could check if it is set and assign accordingly.

The **validate** function is defined to read the password from the file and compare a visitor’s password to the true password. If they are equal then the session variable for being logged in is set to true and they are redirected to **welcome.php**. Otherwise the **$error** flag is set.

Simple Login System

During the running of **login.php**, **$error** is initially **false** and we check if there has been a **POST** submission. If not, we do nothing. If something was **POST**ed, then we run the validation.

The HTML includes a form and an error message if there was a **POST** and the **$error** is flagged as **true**.

Simple Login System

For **welcome.php**, we again start the **’Demo’** session to access **$_SESSION** and its variables.

If either **$_SESSION[’loggedin’]** is not set (so it was never initialized) then clearly the user shouldn’t be there. Or if **$_SESSION[’loggedin’]** is **false**, they also shouldn’t be welcomed. So we render HTML accordingly.

Otherwise, we welcome the visitor.

Simple Login System

When building more complicated pages where there can be multiple things requiring headers, which also includes the **header('Location: where_to_go')**, PHP can throw warnings that the headers have already been sent.

One fix is to make **ob_start();** the first line of PHP code inside the PHP tags. This clears up previous headers and prevents headers from getting sent too early.

Passwords

Passwords should not be stored in their raw form. As a security measure, we can **hash** using a one-directional hashing function into something that looks like absolute gibberish. The general syntax to do so is:

$hashed_value = hash('method', $value);

where **method** can be one of many, many hashing algorithms. Using **'md2'** as one example:

```
// becomes: a9046c73e00331af68917d3804f70655
$hashed_hello = hash('md2', 'hello');
```

We never need to "invert" the mapping. If someone gives us a password, that can be hashed, too, and if it isn't the same as the desired hashed value of the correct password, their password is wrong.

Email

Using PHP we can send emails through the server with the **mail** function.

```php
#!/usr/local/bin/php
<?php
// The message
$message = "This message is going to be sent in the email.\nThis is the second line of the email.";

// Send
mail('foo@bar.com', 'Subject Line', $message);
?>
```

The **mail** function accepts an email address to send to, a subject, and a message body.

Some developers recommend preventing lines of an email from being more than 70 characters by adding:
$message = wordwrap($message, 70, "\r\n");
This inserts a return carriage and new line if a line is "too long". In modern email clients, this is seldom necessary.

HTML Forms

An HTML form can have both **action** and **onsubmit** attributes. When a form is submitted, the script from **onsubmit** is processed first, if anything is given, and it will return **true** if everything is okay and **false** to cancel the submission.

If it returns **true**, the user is redirected to the page specified by **action** with the appropriate method set.

By default, **onsubmit** returns **true** and **action** links back to the page.

HTML Forms

Consider the files

index.html

```html
<!DOCTYPE html>
<html>
<head>
  <title>Main</title>
</head>
<body>
  <form action="other_page.html">
    <input type="submit" value="submit"/>
  </form>

  <form action="write.php" onsubmit="return document.getElementById('x').value.length !== 0;
        " method="post">
    <input type="text" id="x" name="x" />
    <input type="submit" value="submit"/>
  </form>
</body>
</html>
```

write.php

```php
#!/usr/local/bin/php
<?php
  if(isset($_POST['x'])){ // check something posted
    $file = fopen('x.txt', 'w');
    fwrite($file, "${_POST['x']}");
    fclose($file);
  }
?>
```

HTML Forms

The **index.html** has 2 forms. The first form only has a submit button. Whenever it is pressed, the visitor is redirected to **other_page.html** due to the **action** specified.

The second form has a text input and a submit button. When someone tries to submit the form, JavaScript checks if they actually entered something. If they did not, it returns **false** but if they did, it returns **true** and the user is redirected from **action** to **write.php**.

In **write.php**, the **$_POST** is active because the method in the form was **post** and we can write to a file here. Nothing displays on the **write.php** page.

Remark: it would be cleaner to write the JavaScript as a function in another file. But to keep this example as short as possible, it was all written as a single line in the value of **onsubmit**.

PHP Best Coding Practices

Having seen some of the basics of PHP, we can come up with a list of best practices. In addition to all the best coding practices for general programming and JavaScript, including commenting, function documentation, etc., we:

- Use **<?php ... ?>** instead of **<? ... ?>** for robustness.
- Always use single quotes when a string does not need to be parsed.
- Avoid calling functions in loop conditions: initialize bounds outside of the function, for example.
- **echo** multiple things at a time using a comma rather than concatenating strings: this is faster.
- Use the **[]** to add a single element to an array rather than **array_push**.

PHP Best Coding Practices

To elaborate on a couple of those points:

```
// this is bad: count is called repeatedly and unnecessarily!
for($i = 0; $i < count($arr); ++$i) {
   echo $arr[$i] . ’<br/>’; // and this concatenates...
}

// this is good: only call count once
$len = count($arr);
for($i = 0; $i < $len; ++$i) {
   echo $arr[$i], ’<br/>’; // pure echo is faster
}
```

Suppressing Warnings

Sometimes as programmers, we can manage possible failures ourselves but PHP still prints warnings/errors when things go awry.

```php
<?php
$res = fopen('file_that_might_not_exist.txt', 'r');
if($res){ // worked
  // ...
} else { // we KNOW it didn't work and have our own fix!
  // do this
}
?>
```

Above, despite our error management, PHP will still print a warning on the webpage if the file is not found. To silence warnings produced from a function call, we prefix the function name with @:

```php
<?php
  // no warnings will be printed
  $res = @fopen('file_that_might_not_exist.txt', 'r');
?>
```

Superglobals - $_FILES

A web form that accepts files should have the tag structure:

```
<form enctype="multipart/form-data" method="post" ... >
   ...
</form>
```

The method should always be **post** for files and we specifify data are encoded as "multipart/form-data".

Superglobals - $_FILES

Once the form is submitted, the **$_FILES** superglobal stores files that have been uploaded. Suppose we had an **<input type="file" name="paper"** /> in our HTML form. We could then manage it with a PHP script:

```php
$fileName = $_FILES['paper']['name'];
$saveLocation = dirname(realpath(__FILE__)) . '/uploads/' . $fileName;

move_uploaded_file($_FILES['paper']['tmp_name'], $saveLocation);
```

Each uploaded file results in a variable **$_FILES[’its_HTML_form_name’]** with various properties such as ’**name**’, the name the user gave to the file, and ’**tmp_name**’, a temporary directory location given to the uploadedfile.

Superglobals - $_FILES

In specifying **$saveLocation** we are saying we want to save the file to the **uploads** folder that lies within the current working directory of the PHP script. And the file name will be that of **$fileName.**

The **move_uploaded_file** function transfers a file from its temporary location status to the desired directory**.**

Remark: this is kind of like the **mv** command in Linux that can both rename and move a file.

Superglobals - $_FILES

The **uploads** folder should have permissions set to **755.**

Warning: to obtain the current folder path, do not use the **getcwd** PHP function as its behaviour varies across servers. On our severs, this function does not always return the current working directory, which is what it is supposed to return...

JSON

JavaScript Object Notation (JSON) is a means of encoding objects in JavaScript. This format is also conducive to sending information between JavaScript and PHP scripts. A JSON object must always be form the form:

```
{
   "name": value,
   "next_name": next_value
}
```

In this case, all named variables appear in double quotes! The values can be numbers, booleans, strings, null, arrays (with []), or JSON objects.

JSON

Here is an example:

```
{
   "name": "Joe Bruin",
   "age": 20,
   "program": {
      "year": 3,
      "specialization": "computing",
      "courses": ["PIC 10A", "PIC 10B", "PIC 10C", "PIC 40A"]
   }
}
```

Note, despite the name "JavaScript Object Notation", this is not a valid JavaScript object, either, as the properties are given in double-quotes.

AJAX Calling PHP Scripts

An AJAX call can be made to a PHP script. Whatever the PHP script **echo**s is what the AJAX call receives. In the following example, we will add a new customer to our clientele list that is stored in a flat file **clients.txt** with values **Company [TAB] Contact Name [TAB] Email Address.**

We will send the data through an AJAX call with POST; the data will be bundled in a **Client** object.

AJAX Calling PHP Scripts I

index.html

```html
<!-- usual header stuff -->
<form>
  <label for="company">Company:</label> <input type="text" name="company" id="company" /> <br/>
  <label for="contact">Contact Name:</label> <input type="text" name="contact" id="contact" /> <br/>
  <label for="email">Email:</label> <input type="email" name="email" id="email" />  <br/>
  <input type="button" value="Add Contact" onclick="add_client();" />
  <div id="confirm"></div>
</form>
<!-- more HTML -->
```

process.js

```javascript
function Client(company, name, email){
  this.company = company;
  this.name = name;
  this.email = email;
}


function add_client(){
  let company = document.getElementById('company').value;
  let name = document.getElementById('contact').value;
  let email = document.getElementById('email').value;

  let client = new Client(company, name, email);
  let data = JSON.stringify(client);

```

AJAX Calling PHP Scripts II

```
  let xmlhttp = new XMLHttpRequest();
  xmlhttp.onreadystatechange = function() {
    if (this.readyState === 4 && this.status === 200) {
      document.getElementById('confirm').innerHTML = this.responseText;
    }
  };

  xmlhttp.open("POST", "client_post.php", true);
  xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
  xmlhttp.send("client=" + data);
}
```

client_post.php

```
#!/usr/local/bin/php
<?php
  // response type will be text
  header('Content-Type: text/plain; charset=utf-8');

  $client = json_decode($_POST['client'], false);

  $project_space = 100;

  $file = fopen('contacts.txt', 'a');
  if($file){
    fwrite($file, "{$client->company}\t{$client->name}\t{$client->email}\t");
    for($i=0; $i<$project_space;++$i){ // add 100 spaces for 'extra space'
      fwrite($file,' ');
    }
    fwrite($file, "\n"); // go to new line
    fclose($file);
```

AJAX Calling PHP Scripts III

```
    echo 'written';
  }
  else{
    echo 'file fail';
  }
?>
```

AJAX Calling PHP Scripts

Now we can unpack all of this. The boring, easy part is **index.html** that includes the **process.js** script and simply has a web form so that when when the button is pressed, an AJAX call is made that will eventually modify the content of the **div** with ID of "confirm".

The **process.js** has a constructor for a **Client** class that makes an object to represent a client. The **add_client** button collects the information from the form, makes a **Client** and then turns that object into a JSON string by calling **JSON.stringify**.

AJAX Calling PHP Scripts

There is then making an AJAX call. As always, we make an **XMLHttpRequest** object and specify a callback to run when a response is received. In this case we wish to edit the **div.** This time, the **open** function specifies the method as "POST", the script to call as **client_post.php**, and that the call runs asynchronously (**true**), i.e., while the rest of the JS runs, the PHP can run.

Unlike our previous AJAX example, this time we have data to send so we specify what to sent as a query. Recall queries are of the form **name1=value1&name2=value2...**.

Remark: as a rule of thumb based on security and size limits of **GET**, AJAX should be called with **GET** for **idempotent** operations, i.e., operations that retrieve information but to not modify data on the server, and **POST** should be used whenever data are modified or information needs to be more secure.

AJAX Calling PHP Scripts

Within the PHP script, we use the **json_decode** function to convert the **$_POST['client']** string into an object. The boolean argument is used to request an associative array if **true** instead of an object if **false**.

We then open a file and begin writing. Double quotes are used to evaluate the variables. The extra braces **{ ... }** around the properties are used to group **$client->company** as a single expression to be evaluated, for example. For objects in PHP, methods and properties are accessed via **->**.

After writing to the file, the PHP script renders "written" or if there has been an error and the file pointer is NULL then it renders "file fail". Either message is then relayed as the value of **this.responseText** in the AJAX call.

AJAX Calling PHP Scripts

Remarks: HTTP specifies a protocol for how data are sent/received. Before calling **xmlhttp.send**, we specify what is being sent and in what form:

```
xmlhttp.setRequestHeader("Content-type",
"application/x-www-form-urlencoded");
```

This sends a **header**, something storing meta-data to the PHP script that the way content is encoded is through a query string (like with GET).

Likewise, the PHP script adds a header (recall **header**s must be set before any HTML appears) to specify that it is returning plain text.

```
header("Content-Type: text/plain; charset=utf-8");
```

AJAX Calling PHP Scripts

Another remark: updating the flat file to add projects with a client, removing a client, etc. could all become very messy. PHP does support "random access" within a file just as C/C++ with **fseek** but nobody likes working that way...

Ultimately, using a database is a smarter/simpler solution, which will be covered later.

More on JSON

We have seen turning JavaScript objects into JSON with **JSON.stringify** and that PHP can decode JSON strings into arrays/objects with **json_decode**.

It goes the other way, too. We can convert a PHP object into a JSON string with **json_encode.** This will produce a string that in pure JS is interpreted as an array**.**

JavaScript can also parse JSON strings with **JSON.parse**.

Object Oriented Design I

We will give the "equivalent" PHP implementation to the C++ classes below:

```
class Square{
private:
  double length;

protected:
  double get_length() const { return length; }

public:
  Square(double _length) : length(_length) {} // (A)

  double get_area() const {return length*length;} //(B)
};

class ColouredSquare : public Square{ // (C)
```

Object Oriented Design II

```
private:
  std::string colour;

public:
  ColouredSquare(double _length, std::string _colour):
    Square(_length),colour(std::move(_colour)){} //(D)

  void set_colour(std::string _new_colour) {
    colour = std::move(_new_colour);
  }

  void display() const { // (E)
    std::cout << colour + " square of length " <<
      get_length() << " has area" << get_area()
      << '\n';
  }
};
```

Object Oriented Design I

```php
class Square{
  private $length;

  protected function get_length() { return this->length; }

  function __construct($_length) { $this->length = $_length; }

  public function get_area() { return $this->length*$this->length; }
}

class ColouredSquare extends Square{
  private $colour;

  public function __construct($_length, $_colour){
    parent::__construct($_length);
    $this->colour = $_colour;
  }

  public function set_colour($_new_colour){
    $this->colour = $_new_colour;
  }

  public function display(){
    echo "{$this->colour} square of length {$this->get_length()} has area {$this->get_area()}\n";
  }

}
```

Access Modifiers

Like C++, PHP supports **public**, **protected**, and **private** access modifiers. In PHP, they appear right before the method or property. Without listing a modifier, the default behaviour is **public**.

All properties (member variables) must be prefixed by something. If not **public, protected,** or **private**, then the **var** keyword granting public access must be used.

Access Modifiers

Anything that is **public** is accessible to all users of the class and all derived classes. Anything that is **protected** is only accessible within the class or its derived classes. Anything that is **private** is only accessible within the class itself.

Remark: protected should only ever be used for member functions. A property should never be **protected**. Ideally **protected** shouldn’t be used at all.

Constructors and $this

A class constructor (used to make a class object) is a function that must always be named **__construct**. Destructors (what takes place as an object is destroyed) can also be written and they have the name **__destruct**.

The **$this** variable is a reference to the class itself. To access members (properties or methods) of the class, we must use **$this->**.

When accessing a property, we don’t need to prefix that property with a $ again.

Like other functions in PHP, overloading is not possible.

Inheritance

To inherit the properties and methods of another class, PHP uses the **extends** keyword. Hence, **ColouredSquare extends Square**.

To invoke the constructor of the base class, we scope it with **parent::**.

Function Overriding

A derived class can overwrite how a method is implemented. We will write the PHP equivalent of the C++ below:

```
struct Foo{
  virtual void print() const { std::cout << "Foo!"; }
};
struct Bar : Foo{
  void print() const override { std::cout << "Bar!"; }
};

// in code
Foo f; f.print(); // prints Foo!
Bar b; b.print(); // prints Bar!
```

Function Overriding

In PHP:

```
class Foo{
  public function print() { echo 'Foo!'; }
}

class Bar extends Foo{
  public function print() { echo 'Bar!'; }
}

# now we use these
$f = new Foo;
$f->print(); # prints Foo!

$b = new Bar;
$b->print(); # prints Bar!
```

Polymorphism

Unlike C++, PHP will treat function arguments, whether or not they are passed as references/pointers **polymorphically.** The term **polymorphism** in programming referes to being able to handle multiple kinds of objects with a single underlying object type.

We can write a function **manage_Foos** that will only accept objects of type **Foo** or those derived from it.

```php
function manage_Foos(Foo $a_foo){
  $a_foo->print();
}

manage_Foos($f); // prints Foo!
manage_Foos($b); // prints Bar!
manage_Foos(7); // ERROR
```

For classes only, PHP allows us to specify an input data type for a function.

Object Oriented Design

PHP also has **abstract** classes and **interfaces**. As in other languages, they specify what a derived class should do/implement. We won't go into that.

Unlike C++, PHP does not support multiple inheritance. A class **A** cannot extend both **B** and **C**.

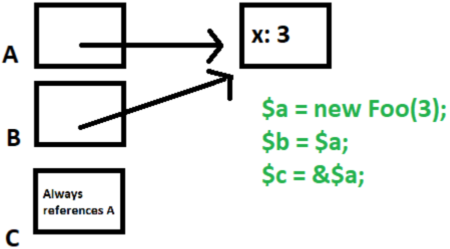
Objects Passed by Reference

In PHP, an object that has been **new**ed is effectively a pointer to the location in memory where the real object lies. Because of this, objects are passed by reference, even if their members are mutated! To force a copy, we use the **clone** keyword (makes a shallow copy) or **deep_copy** function. But we won't get into that.

We will consider a simple snippet of code to illustrate the basic features where objects are passed as references.

Objects Passed by Reference

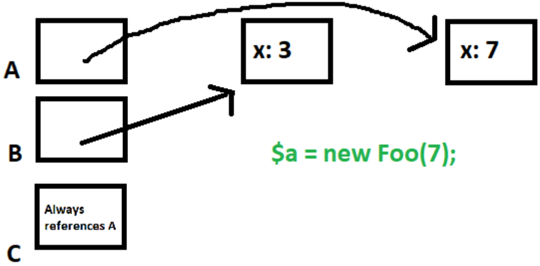
```
<?php
class Foo{
  var $x;
  function __construct($i) { $this->x = $i;}
}

$a = new Foo(3);
$b = $a; // $b and $a both point to same object
$c = &$a;  // $c references the pointer $a
echo "$a->x, $b->x, $c->x<br/>";
$a = new Foo(7); // updates $a and $c but $b stays pointing where it did
echo "$a->x, $b->x, $c->x<br/>";
$c = NULL; // $a and $c are NULL
echo "$b->x<br/>";
?>
```



```
$a = new Foo(3);
$b = $a;
$c = &$a;
```

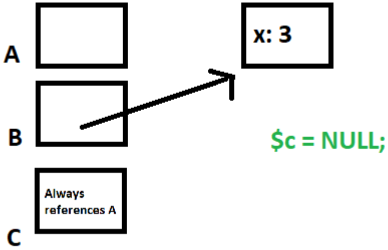
Objects Passed by Reference

```php
<?php
class Foo{
  var $x;
  function __construct($i) { $this->x = $i;}
}

$a = new Foo(3);
$b = $a; // $b and $a both point to same object
$c = &$a;  // $c references the pointer $a
echo "$a->x, $b->x, $c->x<br/>";
$a = new Foo(7); // updates $a and $c but $b stays pointing where it did
echo "$a->x, $b->x, $c->x<br/>";
$c = NULL; // $a and $c are NULL
echo "$b->x<br/>";
?>
```



Objects Passed by Reference

```
<?php
class Foo{
  var $x;
  function __construct($i) { $this->x = $i;}
}

$a = new Foo(3);
$b = $a; // $b and $a both point to same object
$c = &$a;  // $c references the pointer $a
echo "$a->x, $b->x, $c->x<br/>";
$a = new Foo(7); // updates $a and $c but $b stays pointing where it did
echo "$a->x, $b->x, $c->x<br/>";
$c = NULL; // $a and $c are NULL
echo "$b->x<br/>";
?>
```



A

B

C

x: 3

Always references A

$c = NULL;

Objects Passed by Reference

With this understanding, we can consider where a **write on copy** takes place.

```php
<?php
class Foo{
  var $i;
  function __construct($j) { $this->i = $j; }
}

function manip($x){
  ++$x[0]; // modify reference (pointer), $x still references $arr, $a===5
  ++$x[1]->i; // $x[1] reference to $f unchanged, $x still references $arr, $f->i === 2
  $x[1] = new Foo(5); /* $x[1] reference to $f unchanged, $x still references $arr, $f->i === 5 */
  ++$x[2]->i; // $x[2] pointer to object unchanged, $x still references $arr, $g->i === 3

  // $x[2] pointer to object would change here! So now $x is a copy of $arr
  $x[2] = new Foo(8); // $x[2]->i === 8 but $g->i === 3
  ++$x[3]; // $x is a copy of $arr so $b===5 but $x[3] === 6
  ++$x[4]; /* $x is a copy of $arr but $x[4] is a copy of a reference so $x[4] === 7 and $c ===7 */
  ++$x[5]->i; /* $x is a copy of $arr but $x[5] is a copy of a pointer to the object so $h->i === 4 */
}
$f = new Foo(1); $g = new Foo(2); $h = new Foo(3);
$a = 4; $b = 5; $c = 6;
$arr = [&$a, &$f, $g, $b, &$c, $h];
manip($arr);
?>
```

PHP Injecting Code into JavaScript

Here is a short example of PHP injecting code into JS.

```
#!/usr/local/bin/php
<?php
  class Person{
    var $name;
    var $age;
    public function __construct($name, $age){
      $this->name = $name;
      $this->age = $age;
    }
  }
  $michelle = new Person('Michelle', 60); // make PHP object
  $php_json_string = json_encode($michelle); // turn to JSON string
?>
<!DOCTYPE html>
<head>
  <title>PHP to JS</title>
</head>
</body>
  <input type="button" value="Show Info" onclick="alert(michelle.name + ' is ' + michelle.age + ' years old.');" />
  <script> <!-- direct JS -->
    // Parse the output as a string
    let michelle = JSON.parse('<?php echo $php_json_string; ?>');
    // or could just view as an array directly
    // let michelle = <?php echo $php_json_string; ?>;
  </script>
</body>
</html>
```

PHP Injecting Code into JavaScript

The PHP makes all the data. It also renders some **script** tags where in JS, the **michelle** variable is defined through the rendering of PHP.

The pure **$php_json_string** when viewed as JS is an array. But when enclosed in quotes, it is a JSON string to be parsed.

When the button is pressed, we see the alert, "Michelle is 60 years old."

Remark: in some cases such as where PHP needs to make JS, it can be okay to have JS directly on the page. The same goes for when PHP needs to render CSS, which it sometimes does: in these less common cases, CSS may also be defined on the page.

zval

PHP is written in C. In PHP, all variables are of type **zval**, an instance of a class (**struct**) with members:

- **value**, storing the variable's value
- **ref_count**, counting how many variable's reference it
- **type**, storing the variable's type
- **is_ref**, a flag as to whether the variable is a reference

zval

Going just a little more under-the-hood, the **value** is represented as a class (**union**). In C and C++, a **union** is a class that has multiple members but only one can be in a valid state at a given time so the size is based on the size of its largest member. It is a "space saving" class type.

Within the union are found member variables for:

- **long int** (the type of integer value PHP supports)
- **double** (the type of floating value PHP supports)
- **string**-like class (for string literals)
- **hash table**-like class (for an array)
- a handle to return an object's value

var_dump and debug_zval_dump

There is also a quirky function called **debug_zval_dump** which is difficult to use since its implementation has changed a lot over different PHP versions and the answers it outputs depend on internal optimizations of the PHP engine.

Nonetheless, it does illustrate the concept/idea that the variables are represented as **zvals**.

```
$x = ’hello’;
$y = $x;

debug_zval_dump($x); // string(5) "hello" refcount(3)
```

The comment there is the output in PHP 5. In PHP 7, the refcount is 1.

Garbage Collection

PHP uses reference counting to determine when an object should be desctructed.

PHP 5 introduces a destructor concept similar to that of other object-oriented languages, such as C++. The destructor method will be called as soon as there are no other references to a particular object, or in any order during the shutdown sequence. ~PHP Net Manual

Thus, for example, if we define some variables in a function, upon exiting the function, as long as each variable is unreferenced after the function ends, they will be destroyed.

In effect, PHP variables behave like objects referenced through smart pointers in C++, automatically adhering to RAII, provided there's no funny circular references going on.

Memory Leaks

In some cases, since the garbage collector does reference counting, memory leaks can arise!

```php
class Person{
  private $name;
  private $friend;

  function __construct($name) {
    $this->name = $name;
    $this->friend = NULL;
  }

  function __destruct() {
    echo "destroying {$this->name}<br/>";
  }

  function set($friend){
    $this->friend = $friend;
  }
}

function Leaks(){
  $alice = new Person('Alice');
  $bob = new Person('Bob');
  $alice->set($bob);
  $bob->set($alice);
  $colleen = new Person('Colleen'); // has no friends
}

Leaks();

echo 'function call has ended', '<br/>';
```

Memory Leaks

This generates the output:

```
destroying Colleen
function call has ended
destroying Alice
destroying Bob
```

Alice and Bob are only destroyed at the end of the program, not the end of the function. There is a memory leak.

Since a PHP programmer does not have control over memory management, it is the programmer’s responsibility to avoid situations where memory leaks could arise.

Fun remark: Python also has this same flaw and will leak memory in the same situation.