PHP: Constructs and Variables

Introduction

This document describes:

1. the syntax and types of variables,
2. PHP control structures (i.e., conditionals and loops),
3. mixed-mode processing,
4. how to use one script from within another,
5. how to define and use functions,
6. global variables in PHP,
7. special cases for variable types,
8. variable variables,
9. global variables unique to PHP,
10. constants in PHP,
11. arrays (indexed and associative),

Brief overview of variables

The syntax for PHP variables is similar to C and most other programming languages. There are three primary differences:

1. Variable names must be preceded by a dollar sign (`$`).
2. Variables do not need to be declared before being used.
3. Variables are dynamically typed, so you do not need to specify the type (e.g., `int`, `float`, etc.).

Here are the fundamental variable types, which will be covered in more detail later in this document:

- Numeric
  - `integer`. Integers ($\pm 2^{31}$); values outside this range are converted to floating-point.
  - `float`. Floating-point numbers.
  - `boolean`. `true` or `false`; PHP internally resolves these to `1` (one) and `0` (zero) respectively. Also as in C, `0` (zero) is `false` and anything else is `true`.
- `string`. String of characters.
- `array`. An array of values, possibly other arrays. Arrays can be indexed or associative (i.e., a hash map).
- `object`. Similar to a class in C++ or Java. (*NOTE*: Object-oriented PHP programming will not be covered in this course.)
- `resource`. A handle to something that is not PHP data (e.g., image data, database query result).

PHP has a useful function named `var_dump()` that prints the current type and value for one or more variables. Arrays and objects are printed recursively with their values indented to show structure.

```
$a = 35;
$b = "Programming is fun!";
$c = array(1, 1, 2, 3);
var_dump($a,$b,$c);
```

Here's the output from the above code.

```
int(35)
string(19) "Programming is fun!"
array(6) {
  [0]=>
  int(1)
  [1]=>
  int(1)
  [2]=>
  int(2)
  [3]=>
  int(3)
}
```

The variable `$a` is an integer with value `35`. The variable `$b` is a string that contains 19 characters. The variable `$c` is an array with six elements: element zero is an integer whose value is `1`, and so on.

Control structures

The control structures – conditionals and loops – for PHP are nearly identical to C. The following list identifies how PHP's control structure syntax differs from other languages.

- The "else-if" condition is denoted by **`elseif`**. Recall that `else if` is used in C, and `elsif` for Perl.
- Single statements within a condition or loop do not require curly braces, unlike Perl where the braces are mandatory.
- The "cases" within a switch-statement can be strings, unlike C where the cases must be numeric.
- The syntax for the foreach loop is slightly different than Perl. For example, in Perl you would write

```
foreach $val (@array) ...
```

In PHP, you would write

```
foreach ($array as $val) ...
```

Mixed-mode processing

When the PHP interpreter encounters code islands, it switches into parsing mode. This feature is significant for two reasons: you can retain variable scope, and you can distinguish PHP code from markup. Here are two examples that demonstrate these concepts.

```
<?php
  $username = "dknuth";      // Defining a variable in the first code island
?>
...
<h1>Hello World</h1>
<p>Welcome,
<?php
  print "$username";         // Using a variable defined in a previous code island
?>. Enjoy your stay!</p>
```

Even though there is HTML markup between the two code islands, the variable `$username` retains its value. The technical reason for this capability is that the variable `$username` is within the *current file's scope*.

The following example demonstrates how to have specific HTML markup displayed if a given condition is true.

```
<?php
  if ($is_logged_in == true) {
?>
<p>Welcome, member. (<a href="logout.php">Log out</a>)</p>
<p>Check out our new member features below.</p>
<?php
  } else {
?>
<p><a href="register.php">Register for an account</a></p>
<p>You must be a member to view anything on this site, sorry!</p>
<?php
  }
?>
```

The same result could be achieved by using multiple `print` statements within the condition blocks to output the HTML markup.

Including other scripts

As with most programming languages, initialization, function definitions, and common code can be placed in separate files. For example, if you had several constants used by multiple C applications, those constants would be defined in a common header file rather than being duplicated within each source code file. In the case of PHP, these separate scripts typically contain common/shared functions, object definitions, and page layout code.

To include other scripts, use the **`include`** statement.

```
include 'somefile.php';
```

The PHP interpreter essentially "inserts" the contents of the specified file name into the current location. If you try to include a file that does not exist, a warning message will be displayed in the browser.

> *NOTE*: If PHP is configured so that `display_errors` is set to `Off`, the warning will not be seen in the browser. You can, however, see the warning by running the script using the CLI.

Suppose you want to define the heading for each page on your site in the file `pageheader.php`:

```
<?php
  print "<div class=\"pageheader\">Programming is Fun!</div>\n";
  print "<div class=\"commonlinks\">some links would go here</div>\n";
  print "<p>Motto: <i>If programming isn't fun, you're doing it wrong.</i></p>\n";
?>
```

This script can be included by other PHP files. For example, the index page – `index.php` – may look like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>Programming is Fun!</title>
  </head>
  <body>
    <?php
      include 'pageheader.php';
      // More code here
    ?>
  </body>
</html>
```

Potentially each page of your site would use the page header so that (a) you don't have to copy and paste the same header markup into every PHP file that displays HTML content, and (b) if you decide to change the page header content, you only need to modify `pageheader.php`.

The inclusion of other scripts can also be based on a *condition*. Suppose you want to display additional content if the current user is an administrator. (NOTE: This is a contrived example – the variable `$is_admin` has no special meaning.)

```
<?php
  if ($is_admin == true)
    include 'admincontent.php';
?>
```

If there are external scripts that contain code that is mandatory for the current script, use the `require` statement:

```
require 'somefile.php';
```

If a required file does not exist, an error message will be displayed in the browser, and the PHP interpreter will exit (i.e., a fatal error).

NOTE: If PHP is configured so that `display_errors` is set to `Off`, the error message will not be seen in the browser. You can, however, see the error by running the script using the CLI.

If you want to prevent multiple `include`s or `require`s, use the **`include_once`** or **`require_once`** statements. If you attempt to use `include_once` (or `require_once`) on a file that has already been included or required using the "once" functions, that statement will be ignored. Depending on the configuration of the PHP interpreter (i.e., if code caching is enabled), `include_once` and `require_once` prevent multiple compilations of the included/required code, thus decreasing the time required to process your script.

There are a few scenarios where the `include`/`require` and `include_once`/`require_once` constructs can cause confusion. Consider the following examples.

```
include_once 'foo.php';
include 'foo.php';
```

The first statement includes `foo.php` as expected; however, the second statement *also* includes `foo.php`. In other words include does not check to see if the given file has been included using `include_once`.

```
include 'foo.php';
include_once 'foo.php';
```

The second statement will *not* include `foo.php` because it has already been included.

```
function initialize_stuff() {
  include_once 'foo.php';
}
initialize_stuff();
include_once 'foo.php';
```

Suppose `foo.php` initializes some variables. Those variables are initialized within *`initialize_stuff()`'s scope*, meaning that they will be undefined the function returns. Because `foo.php` has already been included, the last `include_once` statement will not be executed, essentially leaving the initialization not performed.

Functions

To define a function, use the **`function`** keyword. For example, function `foo()` below takes no arguments simply returns the integer `1` (one).

```
function foo() {
  return 1;
}
```

As with Perl, functions in PHP do not have a specified return type; therefore, the `return` statement is not required. If you were to use `return` with no argument, the value `NULL` is returned. As with C you can return the results of conditions, in which case `1` (one) is true and `0` (zero) is false. Here is an example:

```
function is_more_than_ten ($i) {
  return $i > 10;
}
```

Arguments are passed by specifying the names within the parentheses of the function definition. Because PHP uses dynamic typing, no data type is necessary. *NOTE*: You can pass arguments by reference (i.e., pointers) as in C; however, that topic is beyond the scope of this document.

```
function display_name_and_age ($name, $age) {
  print "It appears that $name is $age year(s) old.\n";
}
```

You can also specify default values for function arguments.

```
function greet ($name = "user") {
  print "Hello, $name!\n";
}
```

The statement `greet();` displays “Hello, user!”, whereas `greet("Donald");` displays “Hello, Donald!”. If values are passed to the function – rather than being set to a default value – they are used from left to right. Consider the following function:

```
function greet2 ($daypart = "day", $name = "user") {
  print "Good $daypart, $name!\n";
}
```

The statement `greet2();` displays “Good day, user!”, `greet("evening","Donald");` displays “Good evening, Donald!”, and `greet2("Donald");` displays “Good Donald, user!”.

Arguments that have default values should be placed at the *end* of the argument list. For example, the following is **not recommended**:

```
function greet_bad ($daypart = "day", $name) {
  print "Good $daypart, $name!\n";
}
```

The true intent of the statement `greet_bad("Donald")` cannot be inferred. Should the string “Donald” override the default value for `$daypart`, or be assigned to the variable `$name`, which does not have a default value? In this case, `greet_bad("Donald");` displays “Good Donald, !” – where `$daypart` is assigned the value `"Donald"`, and `$name` is undefined, which in the string context is the empty string.

Suppose you have the following function:

```
function foo ($a, $b) { }
```

The statement `foo(10,20,30);` will assign 10 to `$a` and 20 to `$b` – the value 30 is effectively ignored. Because the PHP interpreter does not complain if you specify too many arguments, you can support multiple arguments using the `func_num_args()` and `func_get_arg()` functions. Here is an example function that prints out each of its arguments:

```
function display_arguments () {
  $n = func_num_args();
  for ($i=0; $i < $n; $i++)
    echo "arg ", $i+1, " = ", func_get_arg($i), "\n";
}
```

Global variables

Global variables are stored in a predefined associative array named **`$GLOBALS`**. To assign a value to a global variable, use the following syntax:

```
$GLOBALS['variablename'] = somevalue;
```

To access a global variable, simply provide the variable name as the key to the `$GLOBALS` array. For example,

```
$GLOBALS['theuser'] = "dknuth";     // Assign the value
...
$curr_user = $GLOBALS['theuser'];   // Retrieve the value
```

Alternatively, you can specify that a given variable is global within the current scope using the **`GLOBAL`** keyword. Suppose you have a global variable `$bar` that you want to be updated by function `foo()`.

```
function foo () {
  GLOBAL $bar;
  $bar++;
}
```

In the above example, any use of the variable `$bar` after the `GLOBAL` statement is equivalent to `$GLOBALS['bar']`.

Special cases for variable types

Booleans

As previously mentioned, non-zero numbers are considered to be `true`, and zero-valued numbers (e.g., 0, 0.000) are `false`. Any string with a value is considered to be true, *except* for the empty string (`""`) and `"0"`. *WARNING*: The string `"0.0"` is `true`.

Strings

Strings can be denoted by single (`'`) or double (`"`) quotation marks. The ability to use either type of quotation mark is useful if a string contains one of these marks. As with C you can use the `\'` and `\"` escape codes to explicitly produce the desired quotation mark. The single and double quotation mark delimiters for strings have the same effect as they do in Perl: Characters within single quotation marks are treated as literals, and characters within double quotation marks are interpolated. Interpolation means that values will be substituted for variable names, and escape codes will be replaced with the appropriate characters within the string.

> *NOTE*: To display a dollar sign, use `\$`; otherwise, PHP will try to resolve the character sequence as a variable name.

```
$greeting = 'Hello';
$message1 = "$greeting, world!\n";  // Double quotation marks
$message2 = '$greeting, world!\n';  // Single quotation marks
print "1: $message1\n";
print "2: $message2\n";
```

This is how the output appears before being sent to the browser.

```
1: Hello, world!

2: $greeting, world!\n
```

Note that `$message2` neither contains the value for `$greeting` nor interprets the newline character as calling for a new line.

As in Perl, the concatenation character is the dot (`.`) operator:

```
$a = "Hello, world" . 35 . "!\n";
print $a;
```

This is how the output appears before being sent to the browser.

```
Hello, world35!
```

To handle cases where you need to have text adjacent to a variable name within an interpolated string, use curly-braces distinguish between variable and text.

```
$person_type = "student";
$count = 20;
print "There are $count $person_types.";      // Looks for variable named $person_types
print "There are $count $person_type" . "s";  // Correct, but awkward to read
print "There are $count ${person_type}s";
print "There are $count {$person_type}s";     // Also correct
```

To access individual characters within a string, use the `{x}` notation, where *x* is a zero-based index. As with Perl, you can specify a positive index that is beyond the current length of the string.

```
$foo = "Gleetings";
$foo{1} = "r";
$foo{20} = "!";
print "$foo\n";  // Displays "Greetings           !"
```

Type conversion

PHP is loosely typed, meaning that you can use variables of different types in the same statement. For example, strings that contain valid numeric values can be used in arithmetic expressions.

```
$a = "100";
$b = "324.75";
$sum = $a + $b;
print "$a + $b = $sum";
```

This is how the output may appear in the browser.

```
100 + 324.75 = 424.75
```

With numeric expressions, the following conditions hold:

- strings that start with numerals are cast as numbers (e.g., `"42foo"` is converted to `42`),
- strings that do not contain valid numeric values are assumed to be *zero* (e.g., `"foo42"` is converted to `0`),
- `true` is treated as `1` (one), and `false` is treated as `0` (zero).

Unfortunately PHP does not perform type conversion for arrays (i.e., array-to-string). In other words if you tried to display the values in the array `$myarray` using `print $myarray;`, the output would be `Array`. Displaying boolean values can cause unexpected results as well.

```
$cake_exists = false;
print "The statement, 'The cake is a lie,' is $cake_exists.\n";
```

The above code produces the output, `"The statement, 'The cake is a lie,' is ."`. In the boolean-to-string context, `false` is interpreted as the empty string. If `$cake_exists` were set to `true`, the output would be `"The statement, 'The cake is a lie,' is 1."`.

PHP also supports type casting, using more or less the same syntax as C. Type casting is used to force a particular type, provide extra security, or ensure that a specific type is always used. The following example demonstrates how a floating-point value is truncated when cast as an integer.

```
$a = 100;
$b = 324.75;
$sum = $a + (int) $b;
print "$a + $b = $sum";
```

This is how the output may appear in the browser.

```
100 + 324.75 = 424
```

To address the previous example where boolean values are displayed, consider the following revised code:

```
$cake_exists = false;
print "The statement, 'The cake is a lie,' is " . (int) $cake_exists . ".\n";
```

The above code produces the output, `"The statement, 'The cake is a lie,' is 0."`.

Undefined variables

Undefined variables have a default value of `NULL`. The `NULL` value is interpreted differently depending on the context in which the undefined variable is used (i.e., string, number, etc.). Depending on the configuration of the PHP interpreter, you may or may not see a warning when using an undefined variable. To determine if a variable has been defined, use the **`isset()`** function.

```
$foo = 10;
if (isset($foo) == true)
  print "foo is $foo\n";
else
  print "foo is undefined\n";
if (isset($bar) == true)
  print "bar is $bar\n";
else
  print "bar is undefined\n";
```

The code above will produce the following output:

```
foo is 10
bar is undefined
```

Variable variables

Variable variables allow you to access a variable without using that variable directly. Essentially the first variable contains a string whose value is the name of the second variable (without the dollar sign). The second variable is accessed indirectly by prefixing the first variable with an extra dollar sign. This PHP construct is best described with an example:

```
$val1 = 30;
$val2 = 60;
$foo = "val2";
$bar = $$foo;     // $bar's value is 60 after this statement
```

Superglobals

Superglobals are global variables that a predefined by PHP because they have special uses. The variables themselves are associative arrays. The following table lists the superglobals and their corresponding descriptions.

Superglobal	Description
`$_GET`	Variables sent via an HTTP `GET` request.
`$_POST`	Variables sent via an HTTP `POST` request.
`$_FILES`	Data for HTTP `POST` file uploads.
`$_COOKIE`	Values corresponding to cookies.
`$_REQUEST`	The combination of `$_GET`, `$_POST`, and `$_COOKIE`.
`$_SESSION`	Variables stored in a user's session (server-side data store).
`$_SERVER`	Variables set by the Web server.
`$_ENV`	Environment variables for PHP's host system.
`$GLOBALS`	Global variables (including **`$GLOBALS`**).

The `$_GET`, `$_POST`, `$_FILES`, `$_COOKIE`, and `$_SESSION`, superglobals will be covered in more detail in later documents.

The $_SERVER superglobal has several useful variables.

Variable	Description
HTTP_REFERER	The previous URL (if any).
HTTP_USER_AGENT	Browser name.
PATH_INFO	All characters in the URL after the script name.
PHP_SELF	Current script's file name.
REQUEST_METHOD	Either GET or POST.
QUERY_STRING	All characters after the ? in a GET-based URL.

The HTTP_REFERER and HTTP_USER_AGENT variables should be used with care because their values can be spoofed. The following scenario demonstrates a potential security risk.

> Suppose you have information on your Web site (served from say, programmingisfun.net) that should not be available to the general Web public. One solution is to check the HTTP_REFERER to ensure that programmingisfun.net is contained somewhere in the string. The problem is that the "from" domain can be spoofed by an attacker who knows how to properly modify the hosts file on his/her machine.

The HTTP_USER_AGENT value is most commonly used to modify page layout based on the target browser. For example, if given the same markup and cascading style sheet, Microsoft Internet Explorer and Mozilla Firefox may render the page differently. The solution is to have different markup or style sheets sent to the browser based on the browser's name.

Constants

Constants in PHP are distinct from other variables because no there is no dollar-sign prefix. Constants are also global within the script's scope. To define a constant use the **define()** function:

```
define("constname", somevalue);
```

The constants value can be accessed using its name. However, constants cause a problem for variable variables. To get around this, use the **constant()** function:

```
define("foo", 10);
$bar = "foo"
$baz = constant($bar);
```

PHP has some useful predefined constants.

Constant	Description
__FILE__	Current file name and line of code.
__LINE__	Current line number.
__FUNCTION__	Current function name.
__CLASS__	Current class name.
PHP_EOL	The newline character for the server's operating system.
PHP_VERSION	Current version of PHP.
DEFAULT_INCLUDE_PATH	File paths used if a file is included/required and is not in the current directory.

There are also several mathematical constants, such as M_PI for the floating-point value for pi.

Arrays

PHP has essentially one type of array – the associative array (i.e., hash table). Each element in the array has a *key* and a corresponding *value*. Standard arrays (i.e., indexed arrays) can be used in PHP as well; they are simply associative arrays with integer-indexed keys. We'll begin with indexed arrays to introduce some basic syntax and manipulation functions.

Array basics

There are three ways to populate an array. The first method is to use the `array()` function:

```
$proglangs = array("C", "C++", "Perl", "Java");
$primes    = array(1, 2, 3, 5, 7, 11);
$mixedbag  = array("PHP", 42.8, true);
$emptylist = array();
```

The second method is to access the elements directly using the array operator `[]`:

```
for ($i=0; $i < 10; $i++)
  $nums[$i] = $i+1;
```

Notes about the array operator:

- In Perl you can create *holes* in the array. The above example, which is syntactically identical in Perl, creates a ten-element list with indices ranging from `0` to `9`. The statement `$nums[100] = 101;` will elongate the array, thus making the values for keys `10` through `99` default to `0`. Because indexed arrays are implemented as associative arrays, the statement `$nums[100] = 101;` would *not* yield a larger array: it would yield a five-element array, where the fifth element has a key of `100`.
- Unlike Perl, you cannot use *negative indices*. For example, `$nums[-1]` in Perl will retrieve the last value in the list, whereas PHP will generate an error message.

The third method is to use the array operator with no key provided:

```
for ($i=0; $i < 10; $i++)
  $nums[] = $i+1;
```

This syntax is used less frequently, and has the effect of appending the given value to the array.

As with Perl, arrays are *heterogeneous*, meaning that the data stored in the array does not need to be of the same type. For example, `$mixedbag` contains a string, floating-point value, and a boolean value. To get the number of elements in an array, use the `count()` function.

```
$proglangs = array("C", "C++", "Perl", "Java");
echo "I know ", count($proglangs), " language(s).";   // Displays "I know 4 language(s)."
```

To display the contents of an array, the `print()` function will simply display "`Array`". The `print_r()` function performs a *recursive* output of the given array.

```
$proglangs = array("C", "C++", "Perl", "Java");
print_r($proglangs);
```

Here's the output from the above code:

```
Array
(
[0] => C
[1] => C++
[2] => Perl
[3] => Java
)
```

Notes about array output:

- If you want to use the `print_r()` function for temporary debugging output within the Web browser, surround the `print_r()` call with the HTML `<pre>` and `</pre>` tags to make the output easier to read.
- The `print_r()` function takes an optional second argument, that if set to `true`, will *return* the output as a string instead of printing anything to the console or Web browser. For example, `$res = print_r($proglangs,true);`.
- The `var_dump()` function provides more detail with regard to the individual keys and values stored in the array.

PHP also has a function, `var_export()`, that displays information about an array in PHP syntax.

```
$proglangs = array("C", "C++", "Perl", "Java");
var_export($proglangs);
```

Here's the output from the above code:

```
array (
       0 => 'C',
       1 => 'C++',
       2 => 'Perl',
       3 => 'Java',
)
```

Associative arrays

Associative arrays work much like their indexed counterparts, except that associative arrays can have non-numeric keys (e.g., strings). The same three methods for populating indexed arrays apply, except for the `array()` function:

```
$file_ext = array(".c" => "C",
                  ".cpp" => "C++",
                  ".pl" => "Perl",
                  ".java" => "Java");
```

The value to the left of the `=>` operator is the *key*, and the content after it is the corresponding value.

> *NOTE:* The line breaks between key/value definitions are not required – they provide visual clarity to make the code more readable.

Returning arrays from functions

A common task for functions is to initialize various data structures. Consider the following function (and its call):

```
function init_params () {
    $params["username"] = "dknuth";
    $params["realname"] = "Donald Knuth";
    return $params;
}
...
$params = init_params();
```

The `init_params()` function creates an associative array with two keys, then returns the newly-created array. Suppose the "user name" and "real name" values had to be retrieved from a database, and one or more of the values could not be retrieved. The only way to ensure that the array was correctly populated is to see if the desired keys are defined (i.e., not `NULL`). Now consider the following revision:

```
function init_params (&$params) {
    // Perform a database query here, then populate $params...

    if (query was successful)
        return true;
    return false;
}
...
if (!init_params($params))
    // Display some useful error message here...
```

The syntax becomes slightly more complex because the array is *passed by reference* (denoted by the `&` operator in the function argument list). However, the function can now return a success/failure indicator while populating the array (assuming the query was successful).

Array functions

This section lists common PHP functions that manipulate arrays.

array_diff() -