A guide into PHP for C++ programmers

By Joel Yliluoma, August 2007

Introduction

This document introduces quickly (yet rather thoroughly) the PHP programming language to those who are already well-versed in the C++ programming language (or can read it through experience of other similar languages such as Java). It may also teach new things for those who are already familiar with PHP.

The primary focus is for those who want to use PHP for serious programming, not just for scripting something for web. Also, this page only covers the language; the library functions are not covered.

Table of contents [expand all] [collapse all]

Program structure

PHP is designed originally as a script engine that supports adding script into HTML documents. A typical file might look like this:

```
<h1>Hello</h1>
The time is <?php   print date('Y-m-d H:i:s');   ?>.
<hr>
This is a PHP hypertext processed page.
```

This can be explained by saying that the whole PHP document has an implicit "print" over and around it. The pseudo tag, "`<?php`", starts actual PHP code. From PHP code mode, it is possible to return back to the "print" mode with another pseudo tag, "?>".

If you intend to do actual programming in PHP and not use it as a script insertion engine, you are best off by starting the entire file with:

```
<?php
```

And then just starting coding below it. Example:

Note that it is not necessary to terminate the file with ?>. In fact, if you do, you risk the program outputting extra spaces and newlines if you happen to unknowingly enter them beyond the trailing ?>. This may be important, if you output binary data such as images.

Note: The <?php tag is sometimes written shortly as <?. However, because of possible ambiguities with other types of scripts, it is deprecated to do so.

Main program

In C and C++, the program starts from a magic function called main(). In PHP, there is no such magic function. If you want something executed immediately upon script execution, you just write it outside any function, in the global scope. Example:

```
<?php

function test($who) // declares a function
{
  print "Hello, $who!\n";
}

// this is the main program

$greet = 'world';
test($greet);      // calls the function "test".
```

This example program executes in the following order:

1. Define a function
2. Declare and assign a variable
3. Call a function
4. Terminate

Operators

The same operators that work in C++, work in PHP. However, there are a few new operators:

- === is the same as ==, except that it verifies that the type is identical rather than that they evaluate into a compatible value.
  - Similarly for !== versus !=
- @ suppresses error messages for the expression to which it is prepended.
- . concatenates string values. (And .= as an analogue to +=)
- The meanings of . and -> are different from C++, and there are no unary *, .* and ->* operators.
- The meaning of the unary & operator is different (creates a reference instead of taking the address).

For precedence, you should check the manual.

Comments

PHP supports three styles of comments.

```
/* C style
```

```
# And unix shell/configuration file style.
```

Identifiers

In PHP, all variables begin with a dollar sign: `$`

If you see something like $x somewhere in the code, you definitely know that it is a variable (or in the case of `$this`, acts semantically like a variable).

Anything else, if it consists of letters, may be any of these three:

1. A reserved word (keyword)
2. A user-defined function or method name
3. A class or interface name (only when preceded by `class`, `new`, `extends` or `implements` or followed by `::`)
4. A defined constant

When PHP parses your program, if it encounters an identifier that does not begin with a `$`, it first looks whether it is a reserved word or a constant. If it is neither, the semantics define how it is handled then. If it looks like a function call, it will interpret it as a function call. Otherwise, it will implicitly turn it into a string constant! This can sometimes be a "gotcha" to beginners.

Example:

```
define('test', 9); // define is a reserved word.
                   // Here it defines a constant named "test".

if($a == 5)  // if is a reserved word, $a is a variable
{
  $gruu = 9; // $gruu is a variable
  fun(6);    // fun is a function
  fun(test); // test is a constant, which evaluates into 9
  fun($gruu); // the value of the variable is passed as parameter
  fun(gruu);  // gruu is not a constant, the string literal "gruu" will be assumed
  fun(fun);   // the string literal "fun" will be passed to function fun().
}
```

For brevity, we omit the `<?php` tag in this and all following examples.

The reason why unrecognized identifiers are automatically turned into string literals relies in the callback mechanism. For a function call such as this to work:

```
uasort($myarray, mycomparer);
```

It has to turn "mycomparer" into a string literal. In PHP, if a variable contains a string value and one attempts to call the variable as if it were a function, the function that has the name by that string value will be called.

Constants and literals

Numeric literals

```
$a = 5;       // integer literal 5
$a = 5.3e3;   // floating point literal 5300.0
$a = 0x14;    // integer literal 20
$a = -1;      // integer literal -1
$a = 015;     // integer literal 13 (octal 015)
```

String constructs

String constructs in PHP bear superficial similarity to C++ string literals. However, it is important to recognize the differences.

First of all, there's two ways to declare string constructs.

```
$a = "in double quotes";
$b = 'in single quotes';
```

Both ways can be used interchangeably, but there is a difference in the features supported by each convention.

```
$a = "This string contains a newline, a carriage return and a doublequote.\n \r \"";
$b = 'This string also contains a doublequote, but instead of
      newline, there is a literal backslash and a letter. By
      the way, a string literal may span multiple lines (it
      includes newline literals). " \r \n';
$c = 'String literals may be concatenated with the period operator.' . "\n";
```

- The singlequote-enclosed string supports only two escapes: `\'` and `\\`.
- The doublequote-enclosed string supports more escapes: `\n`, `\r`, `\t`, `\\`, `\$`, `\"`, `\[0-7]{1,3}` (octal character codes) and `\x[0-9A-Fa-f]{1,2}` (hexadecimal character codes).
- All other escapes are literals, where the backslash becomes also part of the string content. (Possible gotcha!)
- The doublequote-enclosed string may also include variables; it is not necessarily a pure literal. The variables are evaluated and coerced into string to construct the string.

```
// These two statements are equivalent:
$a = "Apples are $x each.";
$a = 'Apples are ' . $x . ' each.';

// These two statements are equivalent:
$a = "Apples are \$x each.";
$a = 'Apples are $x each.';

// These three statements are equivalent:
$a = "Your {$item->name} is {$item->kind} and expires in {$item->eta} days.";
$a = "Your ${item->name} is ${item->kind} and expires in ${item->eta} days.";
$a = 'Your ' . $item->name . ' is '
             . $item->kind . ' and expires in '
             . $item->eta . ' days.';

// These four statements are equivalent:
$a = "The file is {$stat["size"]} bytes long.";
$a = "The file is {$stat['size']} bytes long.";
$a = 'The file is ' . $stat["size"] . ' bytes long.';
$a = 'The file is ' . $stat['size'] . ' bytes long.';

// Variable inclusions may be very complex, even recursive,
// to the chagrin of writers of syntax highlighters.
// These two statements are equivalent:
$a = "This is {$table["subtable{$subnums["current"]}"][$position['now']]}.";
```

The author of this article prefers to use the singlequotes whenever the features of the doublequotes are not used in the context, but always the form that requires less use of backslashes.

There is also a third kind of expression that looks a lot like a string constant, but it surrounds the string with backticks, i.e. the grave apostrophe. It parallers the similar syntax of the unix shell.

```
$num_functions = `cat *.php | grep -c function`;
```

This syntax invokes the system shell, passes the command to the shell and returns the output of that shell. This syntax also accepts variable inclusions:

```
$reqtype = 'lines';
$filename = 'test.php';
$linecount = `wc --$reqtype < $filename`;
```

However, it is very dangerous to use unless you know exactly what you are doing; the syntax does not escape parameters automatically, permitting remote code execution in the worst case.

A fourth way of entering string literals exists also. It parallers the here-doc mechanism of the unix shell.

```
$testvar = 1;
$testtext = <<<EOF
This is a long piece of text that may span multiple lines
and contain unescaped 'apostrophes' and "quotes".
As for any other escapes, such as variable references
($testvar) or backslash escapes (\r, \n), it behaves like
the doublequote-enclosed strings.
EOF;
```

Array constructs

Arrays in PHP are constructed very similarly to the way they are constructed in Javascript, Ruby, Python and a few other languages, albeit with more verbose syntax. It also slightly resembles the C++ way, but is more versatile.

```
$powers_of_two = Array(1,2,4,8,16,32,64,128,256,512,1024,2048,4096);
$vg_characters = Array
(
  'supermariobros' => Array(
      'hero' => 'Mario Mario',
      'alternative_hero' => 'Luigi Mario',
      'enemy' => 'King Bowser Koopa',
      'henchmen' => Array('Goomba', 'Lakitu', 'Piranha plant', 'Koopa troopa')
    ),
  'metroid' => Array(
      'hero' => 'Samus Aran',
      'enemy' => 'Metroids'
    ),
  'megaman' => Array(
      'hero' => 'Mega Man',
      'hero_alternative_name' => 'Rockman',
      'enemy' => 'Dr. Wily'
    )
);
```

Note that this is not merely an array initializer. Array constructs can be used in expressions, unlike in C++. It is also not a function call, though it is evaluated at runtime. It is a language construct with its own distinct syntax.

```
function PrintBoxCoordinateList($x,$y, $w,$h)
{
  foreach(Array($x, $x-$w, $x+$w) as $xcoord)
    foreach(Array($y, $y-$h, $y+$h) as $ycoord)
      print "$xcoord,$ycoord\n";
}
```

Arrays are discussed in more detail in the "Data structures" chapter of this document.

Named constants

It is possible to define named constants in PHP. The syntax for them is very different from the way how variables are declared:

```
define('MERCURY_RADIUS', 2440);

echo 'The radius of Mercury is ', MERCURY_RADIUS, " km\n";

$radius = MERCURY_RADIUS;
```

Note that named constants cannot be used in the variable inclusion syntax of doublequote-enclosed strings.

For the syntax of named constants in interfaces and classes, see the section about classes.

Note that the meaning of named constants is different from that in C++: In C++, named constants are simply variables whose value cannot be changed. In PHP, the use is the same but they live in another namespace, and you cannot create a reference to a constant.

Variables, types and scope

Declaration

In PHP, variables do not need to be declared. They exist from the first instant they are referred to. For example, this is valid code:

```
$test_var = 'Testing, testing...';
```

It introduces a variable called $test_var into the current scope and assigns it a string value. If there already exists a variable by that name in the current scope, that variable is changed instead.

Accessing an uninitialized variable (i.e. when the first encounter of the variable is in a reading context) may cause a warning, but it is not an error in PHP.

Scope

The scope of locally declared variables (in functions) is the function. The angle braces `{` and `}` do *not* start a new scope, like they do in C++.

The scope of globally declared variables is the main program. They can be imported to functions using the `global` keyword, but by default, they do not exist in the function scope. This is opposite to how it works in C++.

Example:

```
function test()
{
  $a = 8; // a local variable; global is not affected
  global $a; // now imports the global variable into local scope
  $a = 11; // changes the global variable
}
$a = 9; // changes the global variable
{ $c=4; // declares another global variable
}
print $c; // prints 4; scope did not end with }.
print $b; // prints nothing; there is no initialized
          // variable by that name in this scope.
          // (The actual expression printed is null,
          // which is different from C/C++ null pointer.)
```

Speaking of scope, there is also the *scope resolution operator*, `::`, but unlike in C++, where the scope resolution operator needs to be used only where the symbol is not otherwise found in the current symbol table, in PHP it is always required in the contexts where it can apply (i.e. static class methods and properties).

Types

Variables in PHP are weakly typed. The same variable may denote any type of content depending on the way it is assigned at runtime.

```
$a = 5;            // $a is now an integer
$a = 'test';       // $a is now a string
$a = Array(5,3);   // $a is now an array containing two values
$a = 8.3;          // $a is now a floating point value
$a = new tester;   // $a is now an instance of a class called "tester"
unset($a);         // $a no longer exists. If one tries to read it,
                   // a null value is received. isset($a) will return false.

function test($a) // test is a function that takes a parameter of any type
{
  $a *= 9;
  // If $a is numeric, multiplies it by 9
  // If $a is string, coerces it into integer and then multiplies by 9
  // Otherwise, runtime error
  // Note that $a is an on-write copy of the parameter (see below for references)
}
```

However, it *is* possible to force type constraints in PHP (since PHP 5.1), by the means of "type hinting":

```
function Test(Array $x)
{
  print $x;
}

Test(array(1));   // ok
Test('x');        // error, fails at runtime (can be caught)
```

However, type hinting only works for user-defined classes and Arrays.

```
function Test(int $x)
{
}
Test(5); // Errors, excepts an instance of user-defined class "int".
```

Templates

References

Although PHP supports references, they resemble the C++ pointers more than they resemble the C++ references.

Example code:

```
$a = 5;
$b = 7;
$c = &$a;      // $c is now a reference to $a
$c = 3;        // changes the value of $a
$c = &$b;      // $c is now a reference to $b
$c = 3;        // changes the value of $b
$c = 'test'; // changes the value (and the type) of $b
unset($c);    // now $c no longer exists ($b is not affected)
$c = 9;        // declares and sets the value of $c
```

Function parameters may also be passed by reference. The default is to pass them by value.

```
function test(&$a) // test is a function that takes a reference to a variable
{
  $a = 4;
}

$b = 9;
test($b); // sets $b to 4
test(11); // error, requires a reference to a variable.
```

Note: Passing variables by value is not slow in PHP. PHP uses a lazy algorithm, which only makes a copy if the variable is written to.

Pointers

Pointers are string variables that contain the name of the referred variable or function.

```
$test = 5;
$a = 'test';
$$a = 3;      // changes $test to 3.
```

The PHP manual calls them "variable variables".

The effect can be stacked:

```
$test = 5;
$a = 'test';
$b = 'a';
$$$b = 3; // changes $test to 3.
```

Note however, that variable pointers can only refer to simple variable names.

```
$test = Array(1,2);
$a = 'test[1]';
$$a = 5;  // This does not change the contents of anything in $test.
```

```
$"test" = 5;          // This is not valid code. It is a syntax error.
${'te' . 'st'} = 5; // This however is okay.
```

```
  print "gruu";
}
test();             // This calls test() directly
$a = 'test';
$a();               // This calls test() indirectly

call_user_func($a); // This calls test() indirectly, too
```

Or an object instance:

```
class gruu
{
  public $x;
  function __construct()  { $this->x = 5; }
  function show_x()       { print "x is {$this->x}.\n"; }
  function show_nothing() { print "Nothing.\n"; }
};
$test = new gruu;
$a = 'test';
$$a->x = 5;      // This sets $test->x to 5.
print_r($test); // Displays it.
```

Or a method:

```
$test = new gruu;
$a    = 'show_x';

$test->show_x(); // Calls gruu::show_x() for $test directly.
$test->$a();     // Calls gruu::show_x() for $test indirectly.
```

Or a property:

```
$test = new gruu;
$a    = 'x';

print $test->$a; // prints the value of $test->x.
```

Or an instance and a method simultaneously:

```
$r = Array(&$test, 'show_x');

call_user_func($r); // Calls gruu::show_x() for $test indirectly.
// $r();            // For some reason, this is not accepted by PHP.
```

Or a class name and a method simultaneously:

```
gruu::show_nothing(); // Calls gruu::show_nothing() without an instance, directly.

$r = Array('gruu', 'show_nothing');

call_user_func($r);   // Calls gruu::show_nothing() without an instance, indirectly.
// $r();              // For some reason, this is not accepted by PHP.
```

Memory allocation and lifetime

There are no memory allocation directives in PHP. Basically, everything holds a reference count. When the reference count goes zero, the object referred will be deallocated automatically.

```
{
  static public $num_instances = 0;
  function __construct() { ++ self::$num_instances; }
  function __destruct()  { -- self::$num_instances; }
};

function test()
{
  $a = new Tester;
  $b = new Tester;
  echo "Number of Tester instances: ", Tester::$num_instances, "\n"; // gives 2
  unset($b);
  echo "Number of Tester instances: ", Tester::$num_instances, "\n"; // gives 1
}
echo "Number of Tester instances: ", Tester::$num_instances, "\n"; // gives 0
test();
echo "Number of Tester instances: ", Tester::$num_instances, "\n"; // gives 0
```

Beware: If there are circular references in your code, the memory will never be freed. Example circular reference code (do not use, doing this is bad!):

```
$a = Array(); // create an array
$a[] = &$a;   // $a now contains a reference to itself
unset($a);    // leaks memory!
```

Command structures

If, while and for statements

C and C++:

```
if(condition) statement
if(condition) statement else statement2
if(condition) statement else if(condition2) statement2 else statement3
while(condition) statement
for(expression1; expression2; expression3) statement
do statement while(condition);
```

Note that *statement* in C/C++ has a trailing semicolon, unless it's a *compound statement*, which starts with a { and ends with a }. Because this is a list of syntax, those semicolons are not included; they are assumed to be included in those nonterminals.

PHP:

```
if(condition) statement
if(condition) statement else statement2
if(condition) statement elseif(condition2) statement2 else statement3
while(condition) statement
for(expression1; expression2; expression3) statement
do statement while(condition);
```

As you see, everything is almost the same except for `elseif`, which does not exist in C++. (`else if` is also supported.) The use of semicolons is also the same as in C++.

An alternative syntax exists also, but it is rarely used:

```
for(expression1; expression2; expression3): statements endfor
```

The alternative syntax allows to include multiple statements in the structure body without the use of braces `{` and `}`.

Switch-case statement

The syntax of the switch-case statement is the same in PHP as it is in C++:

```
switch(expression)
{
  case expression1: statement break;
  case expression2: statement break;
  default: statement break;
}
```

However, the expressions tested in the switch-case need not be compile-time constants. In fact, the interpreter interprets each expression sequentially at run-time. Unlike in C++, the PHP switch-case is more optimal than a sequence of `if()` statements only in that it avoids the evaluation of the test expression more than once.

The break statement is optional, as is in C++.

Note that because PHP lacks the concept of labels, the PHP switch-case statement cannot be used to write duff's device. Unlike in C++, the `case`/`default` statement is a rigid part of the switch syntax and not a label.

```
/* This is invalid in PHP, although valid in C&C++ */
switch(expression)
{
  case 1:
    if(condition)
    {
  case 2: // error
      statement
    }
}
```

An alternative syntax exists also, but it is rarely used:

```
switch(expression):
  case expression1: statement break;
  case expression2: statement break;
  default: statement break;
endswitch;
```

Foreach statement

The PHP `foreach` statement is a command structure that provides a way to iterate over the values and/or keys in an array without an explicit for-loop.

```
foreach(array_expression as value_var) statement
foreach(array_expression as &value_var) statement
foreach(array_expression as key_var => value_var) statement
foreach(array_expression as key_var => &value_var) statement
```

Example:

```
}
```

An alternative syntax exists also, but it is rarely used:

```
foreach(array_expression as value_var): statements endforeach
foreach(array_expression as &value_var): statements endforeach
foreach(array_expression as key_var => value_var): statements endforeach
foreach(array_expression as key_var => &value_var): statements endforeach
```

The alternative syntax allows to include multiple statements in the structure body without the use of braces `{` and `}`.

Goto, labels, break and continue

PHP does not support goto or labels.

The `break` and `continue` statements have the almost the same semantics as in C++, except for the following differences:

- An optional number may be added after `break`, telling how many enclosing structures to break out of.
- An optional number may be added after `continue`, telling how many levels of enclosing loops it should skip to the end of.
- A `switch` statement is also considered a looping structure for the purpose of `continue`.

```
// Find a fobrinicatable item:
foreach($items as $v)
{
  if(IsFobrinicatable($v))
    break; //breaks out from the foreach loop
}
```

```
function FeedTroll()
{
  global $trolls, $foods;
  do {
    foreach($trolls as &$troll)
    {
      foreach($foods as &$food)
      {
        if($troll->WantsToEat($food))
        {
          $troll->Eat($food);
          break 3; // exits from three do/foreach constructs at once
        }
      }
    }
    print "Couldn't find a troll to feed.\n";
    return;
  } while(false);
  print "Done feeding.\n";
}
```

Functions

In PHP, functions are identified by their name, and their name only.

- The types of parameters are not specified. (Though can be hinted.)
- The type of return value is not specified.
- Default parameters are possible.
- Parameters may be passed by value or by reference.
- It is not an error to call a function with too many parameters.
- Like in C++, a function call must include parenthesis even if no parameters are passed. (Many built-in commands such as `echo` are exceptions; they are part of the language and not actually functions.)
- Functions cannot be undefined or redefined once defined.
- There are no function prototypes. For PHP to be able to call a function, the definition of the function must exist in any of the code modules parsed so far. If the function is not called, it does not need to be defined.

Examples on function declarations:

```
function test1()
{
  // this function does nothing and returns nothing
}
function test2()
{
  // this function does nothing and returns 5
  return 5;
}
function add($a, $b)
{
  // This function assumes the parameters
  // are of some addable type (integer or float)
  // (Passing strings coerces them into numerals before addition.)
  return $a + $b;
}
function multiply(&$a, $factor=1)
{
  // $a was passed by a reference, $factor by value (and has a default of 1)
  $a *= $factor;
}
function test1($k) { } // this is an error, redefinition of a function
```

Variables declared within the function have the scope of the function.

Variables declared outside the function have the scope outside the function, but *not* inside the function (unless the keyword `global` is used).

Function declarations are always global, even if they are defined within another function.

Examples on function calling:

```
test1(); // calls test1
test2(); // calls test2, ignores return value
$a = test2(); // calls test2, assigns return value into a variable
test1(5,3,1); // calls test1 with extra parameters, not an error
multiply($a);   // multiplies $a by 1 (default parameter)
multiply($a,3); // multiplies $a by 3
multiply(7);    // error, requires a reference to a variable
```

When a function is called with extra parameters, the function may use the standard library function `func_get_args()` and its relatives to acquire the parameters.

there is no separate datatype called function pointer. Instead, string variables are used to dynamically call a function to call. Example:

```
function test()
{
  print "gruu";
}
test();       // This calls test() directly
$a = 'test';
$a();         // This calls test() indirectly
```

This is useful with functions taking callback functions, such as uasort():

```
function mycomparer($a,$b) { return strlen($a) - strlen($b); }
$myarray = Array('dog', 'horse', 'calf');
uasort($myarray, 'mycomparer');
```

More details about pointers are shown in the chapter about pointers.

Anonymous functions (lambda functions)

Although this is not a concept that exists in C++, I thought it would be good to mention it here as well. PHP supports the concept of anonymous functions — i.e. functions that do not have any particular name in the global scope. However, because they are constructed at runtime by parsing the code provided in a string parameter, they are rather clumsy. But they still have their uses:

```
uasort($myarray,
  create_function('$a,$b', 'return strlen($a)-strlen($b);')
      );
```

Because create_function() is just a library function instead of a language construct, if you run this code in a loop, it will parse the function body again and again at each loop, so it has really worse performance compared to named functions. The primary use of create_function() may be to create algorithmically defined functions.

Function-local static variables

Like in C++, the `static` keyword can be used in functions to indicate that the local variable keeps its value over successive calls.

```
function Test()
{
  static $num_calls = 0;
  ++$num_calls;
  print "Number of times Test() has been called: $num_calls\n";
}

Test(); Test(); Test();
```

Data structures (Array)

The principal data structure in PHP is the `Array`. The Array is an associative container, which in C++ terms, resembles mostly the `std::map<>` and `std::vector<>`.

An array is a collection of pairs consisting of the key and the value. The type of the key is an integer, a float or a string. The same array may contain keys of different types. The value may be of any type. The same array may contain values of different types.

```
  5,            // same as 0 => 5
  7,            // same as 1 => 7
  3 => 11,      // key is 3, value is 11
  'x' => 'y',   // key is 'x', value is 'y',
  'moo' => file_get_contents('moo.txt')
                // key is 'moo', value is the return value of that function
                // (evaluated here)
);
$a[1] = 13;     // changes the value corresponding to key 1 to 13
$a['x'] = 'z'; // changes the value corresponding to key 'x' to 'z'
$a[] = 19;      // appends to array extending the largest numeric key,
                // i.e. here it is equivalent to $a[4] = 19;
unset($a['x']);  // erases an array element
$b = &$a[3];    // $b is a reference to the value corresponding to key 3
$a = Array();   // $a is now an empty array.
print $b;       // prints 11 (the reference held prevents
                // the previous array from being deallocated)
unset($b);      // remove variable $b (also decrements the reference count
                // of the previous array to zero, causing its deallocation)
```

To iterate through the values of the array, you can use the `foreach` statement.

Note that the array is not sorted by default. Rather, it retains the order in which it was constructed. There are separate functions for sorting arrays.

Besides the classes (described below), there are no other data structures in PHP. Because the arrays can be nested, they can be used for the purposes of trees and other structures.

Array performance

<to be written>

Modules

Often it is useful to put commonly used code in dedicated files that do not contain a main program; they only contain functions, constants and/or classes. In C++, such files are called header files; in Pascal, they are units. In PHP, they are include files. However, in PHP, because there is no separate concept of a prototype and a definition for a function, the include files always contain also the implementation.

The basic idea is to use `require_once` to include the needed modules. There is also a `require`, which omits the check for multiple inclusion, and `include_once` and `include`, which treat a missing file as a warning instead of an error.

Technically, the require/require_once/include/include_once statements work as if they insert the included file in the exact spot where the inclusion statement was (except for turning the parser into print mode, explaining the need for `<?php`.)

You can also call the inclusion statement from within a function, but note that globals defined within the included file will become locals to that calling function, except for the possible functions and classes it defines.

Here is a basic example of a modular program.

Example main program

```
<?php
require_once 'hellolib.php';
printf("%s\n", hellotext());
```

```
<?php
function hellotext()
{
  return 'Hello, world!';
}
```

Note that the name of the file included may also come from a variable:

```
$modulename = 'testmodule';
if(file_exists("modules/{$modulename}.php"))
{
  include_once ("modules/{$modulename}.php");
  // ... code to use module here
}
```

Classes

PHP supports object-oriented programming that resembles the C++ classes.

Class syntax

```
class ClassName
{
  // This declares some properties:
  public $public_member_var;
  private $private_member_var;

  // This declares the constructor:
  function __construct($param = 3)
  {
    // This sets properties. Note that
    // the $this-> is a mandatory part of the syntax.
    $this->public_member_var = $param;
  }

  // This declares the destructor:
  function __destruct()
  {
  }

  public function PublicMemberFunc()
  {
    // This calls a member function. Note that
    // the $this-> is a mandatory part of the syntax.
    $this->PrivateMemberFunc( $this->private_member_var );

    // This is alternative syntax to call a member function.
    self::PrivateMemberFunc( $this->private_member_var );
  }
  private function PrivateMemberFunc($testparam)
  {
    $this->public_member_var = $testparam;
  }
};
```

Destructor semantics

The destructor of the class will be called when the class instance is destructed — that is, when the last reference to the class instance is removed. See the chapter about memory allocation and lifetime for details.

An instance is created with the new keyword. Access to the class members is through the instance variable and the `->` operator (and in some cases, through the scope resolution operator `::`).

```
$instance = new TestClass;
$instance->one_var = $instance->another_var + 1; // accesses properties
$instance->memberfunction($a, $b); // calls a method.

unset($instance); // Removes reference to the instance,
                  // also destructs the object in this
                  // example because no other references
                  // remain.
```

Use without instance (static methods)

In PHP classes, member methods can be called through an instance or without an instance, much like in C++.

```
class StaticTest
{
  static function ddd()
  {
  }
};

StaticTest::ddd(); // ok
```

It is possible also to call statically methods that are not explicitly defined `static`. However, if the called method accesses instance members through `$this->`, it will fail at runtime.

```
class TestClass
{
  function aaa()
  {
  }
  function bbb()
  {
    $this->ccc();
  }
  function ccc()
  {
  }
};

TestClass::aaa(); // ok
TestClass::bbb(); // error, fails at runtime because $this is not defined.
```

Static properties

Like methods, also properties can be set static. Such properties are common to all instances of the class, instead of every instance owning a copy.

```
class Tester
{
  static public $num_instances = 0;

  function __construct() { ++ self::$num_instances; }
  function __destruct()  { -- self::$num_instances; }
};

$a = new Tester;
```

```
unset($a);

echo "Number of tests remaining:", Tester::$num_instances, "\n"; //displays 1
```

Class constants

Global constants are declared in PHP using the `define` keyword. However, the method to create class constants resembles more the C++ way.

```
class TestClass
{
   public const Gravity = 9.81; // Possible gotcha: Note the lack of $

   function test()
   {
     print self::Gravity; // $this->Gravity does not work here.
   }
};
```

The difference between $this-> and self::

This table explains when it is ok to use `$this->` and when to use `self::`.

Note that `$this->` is merely an alias for $*instance_var*`->` and `self::` is an alias for *class_name*`::`. `parent::` is similarly an alias for *parent_class_name*`::`.

task	$*instance_var*->	*class_name*::	implied
Member properties	MUST	MUST NOT	MUST NOT
Static properties	MUST NOT	MUST	MUST NOT
Member methods	MAY	MAY	MUST NOT
Static methods	MUST NOT	MUST	MUST NOT
Class constants	MUST NOT	MUST	MUST NOT

Here's how you read the table: "To access static methods, you MUST NOT use $*instance_var*`->x`, but you MUST use *class_name*`::x`."

"Implied" stands for the C++ way of simply using the context to deduce whether to call a global function or a member function. Unlike in C++, in PHP you MUST NOT omit the context indicator.

```
class TestClass
{
  public $testprop;

  function Test()
  {
    // This declares a local variable. It does not access
    // the instance property, which would be $this->testprop.
    $testprop = 5;

    // This calls a global function, not $this->Test().
    Test();
  }
};
```

```
{
  public $x;
  function __construct() { $this->x = 'something'; }
  function Moo()
  {
    print "I'm just a regular moo.\n";
    print "My x is {$this->x}.\n";
  }
};
class DerivedClass extends BaseClass
{
  function Moo()
  {
    $this->x = 7;
    print "-- I'm a DerivedClass, and I moo.\n";
    print "However, if I were the BaseClass, I would say:\n";
    parent::Moo(); // calls BaseClass::Moo() explicitly
    // BaseClass::Moo(); // this is also permissible.
  }
};

$tmp1 = new BaseClass;
$tmp1->Moo(); // calls BaseClass::Moo()

$tmp2 = new DerivedClass;
$tmp2->Moo(); // calls Derivedclass::Moo()
```

PHP does not do virtual functions. However, there's little need for that, because PHP does not require the function parameters or the array members to be of any specific type. You can just use the derived class object where you would use the baseclass pointer in C++.

Like in Java, it is possible to define an entire class or some class methods `final`. This indicates that the class, or the particular method, cannot be extended through inheritance, and it will be enforced by the interpreter.

```
final class aaa  // this class may never be extended
{
};
class bbb
{
  final function moo() // this method may never be overloaded
  {
  }
};
```

Abstract methods (i.e. purely virtual methods)

It is possible to declare methods that have no implementation, and which must be overridden in derived classes.

```
class BaseClass
{
  abstract protected function Moo();
};
class DerivedClass extends BaseClass
{
  public function Moo()
  {
    print "Moo!\n";
  }
};
```

There is another way to create abstract methods, through a method called *reflection*. It involves creating an `interface` that describes what a class should implement, and declaring that a class implements that interface.

```
interface GraphicObject
{
  function Draw(&$surface);
  function Clear();
};

class Line implements GraphicObject
{
  private $x1, $y1, $x2, $y2;
  function Clear() { $x1=$y1=$x2=$y2=0; }
};

$n = new Line; // error, Line does not implement Draw().
```

An interface never implements anything; it just shows what should be implemented in order to conform to the interface.

Private and public members

In C++, private and public members are created in a class by the means of sections. In PHP, the publicness/privateness is specified individually for each member function and member variable.

```
class TestClass
{
  public function PublicFunc() { }
  private function PrivateFunc() { }
  public $publicvar;
  private $privatevar;

  protected function ProtectedFunc() { }

  function AlsoPublicFunc() { }  // public is the default.
};
```

The class instance is not a pointer

Even though the instance members are accessed through the `->` operator, which looks like pointer access to C++ users, the instance variable is not a pointer, and it is not a reference. It is an object.

Well, *technically*, internally these things are implemented through a myriad of pointers, but it is not a pointer in the way it would be in C++. You can't change it to point to something else, without it meaning changing the contents of the variable and possibly destructing the class instance in the process. Similarly, the word `new` does not mean memory allocation in this context. It is just part of the syntax that creates instances — it is equivalent to an explicit constructor call in C++.

```
$stringvar = 'aiueo';
$instance  = new TestClass;
printf("%s\n", gettype($stringvar)); // outputs "string"
printf("%s\n", gettype($instance));  // outputs "object"
```

Method pointers

explained in detail in the chapter about Pointers.

Other things of interest

The Classes and Objects section of the PHP manual points out many other features supported by PHP classes, that don't exist in C++. Most importantly perhaps, the magic methods, which will not be documented here for now.

Exceptions

PHP exceptions work in a similar manner to C++ extensions.

```
try
{
  $error = 'Always throw this error';
  throw new Exception($error);

  // Code following an exception is not executed.
  echo 'Never executed';
}
catch (Exception $e)
{
  echo 'Caught exception: ',  $e->getMessage(), "\n";
}

// Continue execution
echo 'Hello World';
```

Useful reading

The official PHP site contains a searchable online manual that is very invaluable in all PHP coding even when you are experienced. This is especially true because the standard library in PHP is quite large, and it is difficult to remember the parameters to all the functions.

- http://www.php.net/
- http://ioreader.com/2007/08/17/11-cool-things-about-php-that-most-people-overlook/

Last edited at: 2010-02-16T23:18:13+02:00