




- [Downloads](#)
- [Documentation](#)
- [Get Involved](#)
- [Help](#)
- 

[Dutch PHP Conference 2024](#)

[Getting Started](#)

- [Introduction](#)
- [A simple tutorial](#)

[Language Reference](#)

- [Basic syntax](#)
- [Types](#)
- [Variables](#)
- [Constants](#)
- [Expressions](#)
- [Operators](#)
- [Control Structures](#)
- [Functions](#)
- [Classes and Objects](#)
- [Namespaces](#)
- [Enumerations](#)
- [Errors](#)
- [Exceptions](#)
- [Fibers](#)
- [Generators](#)
- [Attributes](#)
- [References Explained](#)
- [Predefined Variables](#)
- [Predefined Exceptions](#)
- [Predefined Interfaces and Classes](#)
- [Predefined Attributes](#)
- [Context options and parameters](#)
- [Supported Protocols and Wrappers](#)

[Security](#)

- [Introduction](#)
- [General considerations](#)
- [Installed as CGI binary](#)
- [Installed as an Apache module](#)
- [Session Security](#)
- [Filesystem Security](#)
- [Database Security](#)
- [Error Reporting](#)
- [User Submitted Data](#)
- [Hiding PHP](#)
- [Keeping Current](#)

[Features](#)

- [HTTP authentication with PHP](#)
- [Cookies](#)
- [Sessions](#)
- [Dealing with XForms](#)
- [Handling file uploads](#)
- [Using remote files](#)
- [Connection handling](#)
- [Persistent Database Connections](#)
- [Command line usage](#)
- [Garbage Collection](#)
- [DTrace Dynamic Tracing](#)

[Function Reference](#)

- [Affecting PHP's Behaviour](#)
- [Audio Formats Manipulation](#)
- [Authentication Services](#)
- [Command Line Specific Extensions](#)
- [Compression and Archive Extensions](#)
- [Cryptography Extensions](#)
- [Database Extensions](#)
- [Date and Time Related Extensions](#)
- [File System Related Extensions](#)
- [Human Language and Character Encoding Support](#)
- [Image Processing and Generation](#)
- [Mail Related Extensions](#)
- [Mathematical Extensions](#)
- [Non-Text MIME Output](#)

- [Process Control Extensions](#)
- [Other Basic Extensions](#)
- [Other Services](#)
- [Search Engine Extensions](#)
- [Server Specific Extensions](#)
- [Session Extensions](#)
- [Text Processing](#)
- [Variable and Type Related Extensions](#)
- [Web Services](#)
- [Windows Only Extensions](#)
- [XML Manipulation](#)
- [GUI Extensions](#)

Keyboard Shortcuts

?	This help
j	Next menu item
k	Previous menu item
g p	Previous man page
g n	Next man page
G	Scroll to bottom
g g	Scroll to top
g h	Goto homepage
g s	Goto search (current page)
/	Focus search box

[Возврат значений »](#)
[« Функции, определяемые пользователем](#)

- [Руководство по PHP](#)
- [Справочник языка](#)
- [Функции](#)

Change language:

Russian

[Submit a Pull Request](#) [Report a Bug](#)

Аргументы функции

Функция может принимать информацию в виде списка аргументов, который является списком разделённых запятыми выражений. Аргументы вычисляются слева направо перед фактическим вызовом функции (*энергичное* вычисление).

PHP поддерживает передачу аргументов по значению (по умолчанию), [передачу аргументов по ссылке](#), и [значения по умолчанию](#). [Списки аргументов переменной длины](#) и [именованные аргументы](#) также поддерживаются.

Пример #1 Передача массива в функцию

```
<?php
function takes_array($input)
{
echo "$input[0] + $input[1] = ", $input[0]+$input[1];
}
?>
```

Начиная с PHP 8.0.0, список аргументов функции может содержать завершающую запятую, которая будет проигнорирована. Это полезно в случае, когда список аргументов очень длинный, либо если имена переменных длинные, что подталкивает к их вертикальному расположению.

Пример #2 Список аргументов функции с завершающей запятой

```
<?php
function takes_many_args(
    $first_arg,
    $second_arg,
    $a_very_long_argument_name,
    $arg_with_default = 5,
    $again = 'a default string', // Эта завершающая запятая допустима только начиная с 8.0.0.
)
{
    // ...
}
?>
```

Передача аргументов по ссылке

По умолчанию аргументы в функцию передаются по значению (это означает, что если вы измените значение аргумента внутри функции, то вне её значение всё равно останется прежним). Если вы хотите разрешить функции модифицировать свои аргументы, вы должны передавать их по ссылке.

Если вы хотите, чтобы аргумент всегда передавался по ссылке, вы можете указать амперсанд (&) перед именем аргумента в описании функции:

Пример #3 Передача аргументов по ссылке

```
<?php
function add_some_extra(&$string)
{
    $string .= 'и кое-что ещё.';
}
$str = 'Это строка, ';
add_some_extra($str);
echo $str; // выведет 'Это строка, и кое-что ещё.'
?>
```

Передача значения в качестве аргумента, которое должно передаваться по ссылке, является ошибкой.

Значения аргументов по умолчанию

Функция может определять значения по умолчанию для аргументов, используя синтаксис, подобный присвоению переменной. Значение по умолчанию используется только в том случае, если параметр не указан; в частности, обратите внимание, что передача `null` не присваивает значение по умолчанию.

Пример #4 Использование значений по умолчанию в определении функции

```
<?php
function makecoffee($type = "капучино")
{
    return "Готовим чашку $type.\n";
}
echo makecoffee();
echo makecoffee(null);
echo makecoffee("эспрессо");
?>
```

Результат выполнения приведённого примера:

```
Готовим чашку капучино.
Готовим чашку .
Готовим чашку эспрессо.
```

Значениями параметров по умолчанию могут быть скалярные значения, массивы (array), специальный тип `null`, и, начиная с версии PHP 8.1.0, объекты, использующие синтаксис [new ClassName\(\)](#).

Пример #5 Использование нескалярных типов в качестве значений по умолчанию

```
<?php
function makecoffee($types = array("капучино"), $coffeeMaker = NULL)
{
    $device = is_null($coffeeMaker) ? "вручную" : $coffeeMaker;
    return "Готовлю чашку ".join(" ", $types)." $device.\n";
}
echo makecoffee();
echo makecoffee(array("капучино", "лавацца"), "в чайнике");
?>
```

Пример #6 Использование объектов в качестве значений по умолчанию (начиная с PHP 8.1.0)

```
<?php
class DefaultCoffeeMaker {
    public function brew() {
        return 'Приготовление кофе.';
    }
}
class FancyCoffeeMaker {
    public function brew() {
        return 'Приготовление прекрасного кофе специально для вас.';
    }
}
function makecoffee($coffeeMaker = new DefaultCoffeeMaker)
{
    return $coffeeMaker->brew();
}
echo makecoffee();
echo makecoffee(new FancyCoffeeMaker);
?>
```

Значение по умолчанию должно быть константным выражением, а не (к примеру) переменной или вызовом функции/метода класса.

Обратите внимание, что любые необязательные аргументы должны быть указаны после любых обязательных аргументов, иначе они не могут быть опущены при вызове. Рассмотрим следующий пример:

Пример #7 Некорректное использование значений по умолчанию

```
<?php
function makeyogurt($container = "миска", $flavour)
{
    return "Делаем $container с $flavour йогуртом.\n";
}

echo makeyogurt("малиновым"); // "малиновым" - это $container, не $flavour
?>
```

Результат выполнения приведённого примера:

Fatal error: Uncaught ArgumentCountError: Too few arguments to function makeyogurt(), 1 passed in example.php on line 42

Теперь сравним его со следующим примером:

Пример #8 Корректное использование значений по умолчанию

```
<?php
function makeyogurt($flavour, $container = "миска")
{
    return "Делаем $container с $flavour йогуртом.\n";
}

echo makeyogurt("малиновым"); // "малиновым" - это $flavour
?>
```

Результат выполнения приведённого примера:

Делаем миску с малиновым йогуртом.

Начиная с PHP 8.0.0, [именованные аргументы](#) можно использовать для пропуска нескольких необязательных параметров.

Пример #9 Правильное использование аргументов функций по умолчанию

```
<?php
function makeyogurt($container = "миска", $flavour = "малиновым", $style = "греческим")
{
    return "Делаем $container с $flavour $style йогуртом.\n";
}
echo makeyogurt(style: "натуральным");
?>
```

Результат выполнения приведённого примера:

Делаем миску с малиновым натуральным йогуртом.

Начиная с PHP 8.0.0, объявление обязательных аргументов после необязательных аргументов является *устаревшим*. Обычно это можно решить отказавшись от значения по умолчанию, поскольку оно никогда не будет использоваться. Исключением из этого правила являются аргументы вида `Type $param = null`, где `null` по умолчанию делает тип неявно обнуляемым. Такое использование остаётся допустимым, хотя рекомендуется использовать явный [тип nullable](#).

Пример #10 Объявление необязательных аргументов после обязательных аргументов

```
<?php
function foo($a = [], $b) {} // По умолчанию не используется; устарел, начиная с версии PHP 8.0.0
function foo($a, $b) {} // Функционально эквивалентны, без уведомления об устаревании
function bar(A $a = null, $b) {} // Все еще разрешено; $a является обязательным, но допускающим значение null
function bar(?A $a, $b) {} // Рекомендуется
?>
```

Замечание: Начиная с PHP 7.1.0, опущение параметра, не заданного по умолчанию, выбрасывает исключение [ArgumentCountError](#); в предыдущих версиях это вызывало предупреждение.

Замечание: Значения по умолчанию могут быть переданы по ссылке.

Списки аргументов переменной длины

PHP поддерживает списки аргументов переменной длины для функций, определяемых пользователем с помощью добавления многоточия (...).

Список аргументов может содержать многоточие (...), чтобы показать, что функция принимает переменное количество аргументов. Аргументы в этом случае будут переданы в виде массива:

Пример #11 Использование ... для доступа к аргументам

```
<?php
function sum(...$numbers) {
    $acc = 0;
    foreach ($numbers as $n) {
        $acc += $n;
    }
    return $acc;
}

echo sum(1, 2, 3, 4);
```

```
?>
```

Результат выполнения приведённого примера:

```
10
```

Многоточие (...) можно использовать при вызове функции, чтобы распаковать массив (array) или [Traversable](#) переменную в список аргументов:

Пример #12 Использование ... для передачи аргументов

```
<?php
function add($a, $b) {
    return $a + $b;
}

echo add(...[1, 2])."\n";

$a = [1, 2];
echo add(...$a);

?>
```

Результат выполнения приведённого примера:

```
3
3
```

Можно задать несколько аргументов в привычном виде, а затем добавить В этом случае ... поместит в массив только те аргументы, которые не нашли соответствия указанным в объявлении функции.

Также можно добавить [объявление типа](#) перед В этом случае все аргументы, обработанные многоточием (...), должны соответствовать этому типу параметра.

Пример #13 Аргументы с подсказкой типа

```
<?php
function total_intervals($unit, DateInterval ...$intervals) {
    $time = 0;
    foreach ($intervals as $interval) {
        $time += $interval->$unit;
    }
    return $time;
}

$a = new DateInterval('P1D');
$b = new DateInterval('P2D');
echo total_intervals('d', $a, $b).' days';

// Это не работает, т.к. null не является объектом DateInterval.
echo total_intervals('d', null);

?>
```

Результат выполнения приведённого примера:

```
3 days
Catchable fatal error: Argument 2 passed to total_intervals() must be an instance of DateInterval, null given, called in - on line 14 and defined in - on line 2
```

В конце концов, можно передавать аргументы [по ссылке](#). Для этого перед ... нужно поставить амперсанд (&).

Именованные аргументы

В PHP 8.0.0 в виде продолжения позиционных параметров появились именованные аргументы. С их помощью аргументы функции можно передавать по имени параметра, а не по его позиции. Таким образом аргумент становится самодокументированным, независимым от порядка и указанного значения по умолчанию.

Именованные аргументы передаются путём добавления через двоеточия имени параметра перед его значением. В качестве имён параметров можно использовать зарезервированные ключевые слова. Имя параметра должно быть идентификатором, т.е. он не может быть создан динамически.

Пример #14 Синтаксис именованного аргумента

```
<?php
myFunction(paramName: $value);
array_foobar(array: $value);

// НЕ поддерживается.
function_name($variableStoringParamName: $value);

?>
```

Пример #15 Позиционные аргументы в сравнении с именованными аргументами

```
<?php
// Использование позиционных аргументов:
array_fill(0, 100, 50);

// Использование именованных аргументов:
array_fill(start_index: 0, count: 100, value: 50);

?>
```

Порядок, в котором передаются именованные аргументы, не имеет значения.

Пример #16 Тот же пример, что и выше, но с другим порядком параметров

```
<?php
array_fill(value: 50, count: 100, start_index: 0);
?>
```

Именованные аргументы можно комбинировать с позиционными. В этом случае именованные аргументы должны следовать после позиционных аргументов. Также возможно передать только часть необязательных аргументов функции, независимо от их порядка.

Пример #17 Объединение именованных аргументов с позиционными аргументами

```
<?php
htmlspecialchars($string, double_encode: false);
// То же самое
htmlspecialchars($string, ENT_QUOTES | ENT_SUBSTITUTE | ENT_HTML401, 'UTF-8', false);
?>
```

Передача одного и того же параметра несколько раз приводит к выбрасыванию исключения Еггог.

Пример #18 Ошибка, возникающая при передаче одного и того же параметра несколько раз

```
<?php
function foo($param) { ... }

foo(param: 1, param: 2);
// Error: Named parameter $param overwrites previous argument
foo(1, param: 2);
// Error: Named parameter $param overwrites previous argument
?>
```

Начиная с PHP 8.1.0, можно использовать именованные аргументы после распаковки аргументов. Именованный аргумент *не должен* переопределять уже распакованный аргумент.

Пример #19 Пример использования именованных аргументов после распаковки

```
<?php
function foo($a, $b, $c = 3, $d = 4) {
    return $a + $b + $c + $d;
}
var_dump(foo(...[1, 2], d: 40)); // 46
var_dump(foo(...['b' => 2, 'a' => 1], d: 40)); // 46
var_dump(foo(...[1, 2], b: 20)); // Фатальная ошибка. Именованный аргумент $b переопределяет предыдущий аргумент
?>
```

[+ add a note](#)

User Contributed Notes 14 notes

[up](#)
[down](#)
121
[php at richardneill dot org ¶](#)
8 years ago

To experiment on performance of pass-by-reference and pass-by-value, I used this script. Conclusions are below.

```
#!/usr/bin/php
<?php
function sum($array,$max){ //For Reference, use: "&$array"
$sum=0;
for ($i=0; $i<2; $i++){
#$array[$i]++; //Uncomment this line to modify the array within the function.
$sum += $array[$i];
}
return ($sum);
}

$max = 1E7 //10 M data points.
$data = range(0,$max,1);

$start = microtime(true);
for ($x = 0 ; $x < 100; $x++){
$sum = sum($data, $max);
}
$end = microtime(true);
echo "Time: " . ($end - $start) . " s\n";

/* Run times:
# PASS BY MODIFIED? Time
- -----
1 value no 56 us
2 reference no 58 us
```

3 value yes 129 s
4 reference yes 66 us

Conclusions:

1. PHP is already smart about zero-copy / copy-on-write. A function call does NOT copy the data unless it needs to; the data is only copied on write. That's why #1 and #2 take similar times, whereas #3 takes 2 million times longer than #4.
[You never need to use &\$array to ask the compiler to do a zero-copy optimisation; it can work that out for itself.]

2. You do use &\$array to tell the compiler "it is OK for the function to over-write my argument in place, I don't need the original any more." This can make a huge difference to performance when we have large amounts of memory to copy.
(This is the only way it is done in C, arrays are always passed as pointers)

3. The other use of & is as a way to specify where data should be *returned*. (e.g. as used by exec()).
(This is a C-like way of passing pointers for outputs, whereas PHP functions normally return complex types, or multiple answers in an array)

4. It's unhelpful that only the function definition has &. The caller should have it, at least as syntactic sugar. Otherwise it leads to unreadable code: because the person reading the function call doesn't expect it to pass by reference. At the moment, it's necessary to write a by-reference function call with a comment, thus:
\$sum = sum(\$data,\$max); //warning, \$data passed by reference, and may be modified.

5. Sometimes, pass by reference could be at the choice of the caller, NOT the function definition. PHP doesn't allow it, but it would be meaningful for the caller to decide to pass data in as a reference. i.e. "I'm done with the variable, it's OK to stomp on it in memory".

*/
?>
[up](#)
[down](#)

3
[Simmo at 9000 dot 000 ¶](#)
1 year ago

For anyone just getting started with php or searching, for an understanding, on what this page describes as a "... token" in Variable-length arguments:
<https://www.php.net/manual/en/functions.arguments.php#functions.variable-arg-list>
<?php

```
func($a, ...$b)
```

?>
The 3 dots, or elipsis, or "...", or dot dot dot is sometimes called the "spread operator" in other languages.

As this is only used in function arguments, it is probably not technically an true operator in PHP. (As of 8.1 at least?).

(With having an difficult to search for name like "... token", I hope this note helps someone).

[up](#)
[down](#)
15
[LilyWhite ¶](#)

2 years ago
It is worth noting that you can use functions as function arguments

```
<?php
function run($op, $a, $b) {
return $op($a, $b);
}
```

```
$add = function($a, $b) {
return $a + $b;
};
```

```
$mul = function($a, $b) {
return $a * $b;
};
```

```
echo run($add, 1, 2), "\n";
echo run($mul, 1, 2);
?>
```

Output:

3
2
[up](#)
[down](#)

2
[tianyiw at vip dot qq dot com ¶](#)
1 year ago

```
<?php
/**
 * Create an array using Named Parameters.
 *

```

```
* @param mixed ...$values
* @return array
*/
function arr(mixed ...$values): array
{
    return $values;
}

$arr = arr(
    name: 'php',
    mobile: 123456,
);
```

```
var_dump($arr);
// array(2) {
//   ["name"]=>
//   string(3) "php"
//   ["mobile"]=>
//   int(123456)
// }
```

[up](#)
[down](#)

30
[gabriel at figdice dot org](#)
7 years ago

A function's argument that is an object, will have its properties modified by the function although you don't need to pass it by reference.

```
<?php
$x = new stdClass();
$x->prop = 1;

function f ( $o ) // Notice the absence of &
{
    $o->prop ++;
}

f($x);

echo $x->prop; // shows: 2
?>
```

This is different for arrays:

```
<?php
$y = [ 'prop' => 1 ];

function g( $a )
{
    $a['prop'] ++;
    echo $a['prop']; // shows: 2
}

g($y);

echo $y['prop']; // shows: 1
?>
```

[up](#)
[down](#)

14
[Hayley Watson](#)
6 years ago

There are fewer restrictions on using ... to supply multiple arguments to a function call than there are on using it to declare a variadic parameter in the function declaration. In particular, it can be used more than once to unpack arguments, provided that all such uses come after any positional arguments.

```
<?php

$array1 = [[1],[2],[3]];
$array2 = [4];
$array3 = [5],[6],[7];

$result = array_merge(...$array1); // Legal, of course: $result == [1,2,3];
$result = array_merge($array2, ...$array1); // $result == [4,1,2,3]
$result = array_merge(...$array1, $array2); // Fatal error: Cannot use positional argument after argument unpacking.
$result = array_merge(...$array1, ...$array3); // Legal! $result == [1,2,3,5,6,7]
?>
```

The Right Thing for the error case above would be for \$result==[1,2,3,4], but this isn't yet (v7.1.8) supported.

[up](#)
[down](#)

12

[boan dot web at outlook dot com ¶](#)

6 years ago

Quote:

"The declaration can be made to accept NULL values if the default value of the parameter is set to NULL."

But you can do this (PHP 7.1+):

```
<?php
function foo(?string $bar) {
    //...
}

foo(); // Fatal error
foo(null); // Okay
foo('Hello world'); // Okay
?>
```

[up](#)
[down](#)

4

[Luna ¶](#)

1 year ago

When using named arguments and adding default values only to some of the arguments, the arguments with default values must be specified at the end or otherwise PHP throws an error:

```
<?php

function test1($a, $c, $b = 2)
{
    return $a + $b + $c;
}

function test2($a, $b = 2, $c)
{
    return $a + $b + $c;
}

echo test1(a: 1, c: 3)."\n"; // Works
echo test2(a: 1, c: 3)."\n"; // ArgumentCountError: Argument #2 ($b) not passed

?>
```

I assume that this happens because internally PHP rewrites the calls to something like test1(1, 3) and test2(1, , 3). The first call is valid, but the second obviously isn't.

[up](#)
[down](#)

12

[Sergio Santana: ssantana at tlaloc dot imta dot mx ¶](#)

18 years ago

PASSING A "VARIABLE-LENGTH ARGUMENT LIST OF REFERENCES" TO A FUNCTION

As of PHP 5, Call-time pass-by-reference has been deprecated, this represents no problem in most cases, since instead of calling a function like this:

```
myfunction($arg1, &$amp;arg2, &$amp;arg3);
```

you can call it

```
myfunction($arg1, $arg2, $arg3);
```

provided you have defined your function as

```
function myfunction($a1, &$amp;a2, &$amp;a3) { // so &$amp;a2 and &$amp;a3 are
// declared to be refs.
... <function-code>
}
```

However, what happens if you wanted to pass an undefined number of references, i.e., something like:

```
myfunction(&$arg1, &$amp;arg2, ..., &$amp;arg-n);?
```

This doesn't work in PHP 5 anymore.

In the following code I tried to amend this by using the array() language-construct as the actual argument in the call to the function.

```
<?php

function aa ($A) {
    // This function increments each
    // "pseudo-argument" by 2s
    foreach ($A as &$amp;x) {
        $x += 2;
    }
}
```

```
$x = 1; $y = 2; $z = 3;
```

```
aa(array(&$x, &$y, &$z));
echo "---$x--$y--$z--\n";
// This will output:
// --3--4--5--
?>
```

I hope this is useful.

Sergio.

[up](#)
[down](#)

11

[jcaplan at bogus dot amazon dot com ¶](#)

17 years ago

In function calls, PHP clearly distinguishes between missing arguments and present but empty arguments. Thus:

```
<?php
function f( $x = 4 ) { echo $x . "\n"; }
f(); // prints 4
f( null ); // prints blank line
f( $y ); // $y undefined, prints blank line
?>
```

The utility of the optional argument feature is thus somewhat diminished. Suppose you want to call the function `f` many times from function `g`, allowing the caller of `g` to specify if `f` should be called with a specific value or with its default value:

```
<?php
function f( $x = 4 ) {echo $x . "\n"; }

// option 1: cut and paste the default value from f's interface into g's
function g( $x = 4 ) { f( $x ); f( $x ); }

// option 2: branch based on input to g
function g( $x = null ) { if ( !isset( $x ) ) { f(); f(); } else { f( $x ); f( $x ); } }
?>
```

Both options suck.

The best approach, it seems to me, is to always use a sentinel like `null` as the default value of an optional argument. This way, callers like `g` and `g`'s clients have many options, and furthermore, callers always know how to omit arguments so they can omit one in the middle of the parameter list.

```
<?php
function f( $x = null ) { if ( !isset( $x ) ) $x = 4; echo $x . "\n"; }

function g( $x = null ) { f( $x ); f( $x ); }
```

```
f(); // prints 4
f( null ); // prints 4
f( $y ); // $y undefined, prints 4
g(); // prints 4 twice
g( null ); // prints 4 twice
g( 5 ); // prints 5 twice
```

?>

[up](#)
[down](#)

4

[catman at esteticas dot se ¶](#)

8 years ago

I wondered if variable length argument lists and references works together, and what the syntax might be. It is not mentioned explicitly yet in the php manual as far as I can find. But other sources mention the following syntax "`&...$variable`" that works in php 5.6.16.

```
<?php
function foo(&...$args)
{
    $i = 0;
    foreach ($args as &$arg) {
        $arg = ++$i;
    }
}
foo($a, $b, $c);
echo 'a = ', $a, ', b = ', $b, ', c = ', $c;
?>
```

Gives

```
a = 1, b = 2, c = 3
```

[up](#)
[down](#)

3

6 years ago

If you use ... in a function's parameter list, you can use it only once for obvious reasons. Less obvious is that it has to be on the LAST parameter; as the manual puts it: "You may specify normal positional arguments BEFORE the ... token. (emphasis mine).

```
<?php
function variadic($first, ...$most, $last)
{ /*etc.*/ }
```

variadic(1, 2, 3, 4, 5);
?>
results in a fatal error, even though it looks like the Thing To Do™ would be to set \$first to 1, \$most to [2, 3, 4], and \$last to 5.

up
down

1
info at keraweb dot nl

6 years ago

You can use a class constant as a default parameter.

```
<?php

class A {
const F00 = 'default';
function bar( $val = self::F00 ) {
echo $val;
}
}
```

\$a = new A();
\$a->bar(); // Will echo "default"

up
down

2
John

17 years ago

This might be documented somewhere OR obvious to most, but when passing an argument by reference (as of PHP 5.04) you can assign a value to an argument variable in the function call. For example:

```
function my_function($arg1, &$arg2) {
if ($arg1 == true) {
$arg2 = true;
}
}
my_function(true, $arg2 = false);
echo $arg2;
```

outputs 1 (true)

my_function(false, \$arg2 = false);
echo \$arg2;

outputs 0 (false)

+add a note

- [Функции](#)
 - [Функции, определяемые пользователем](#)
 - [Аргументы функции](#)
 - [Возврат значений](#)
 - [Обращение к функциям через переменные](#)
 - [Встроенные функции](#)
 - [Анонимные функции](#)
 - [Стрелочные функции](#)
 - [Синтаксис callable-объектов первого класса](#)

- [Copyright © 2001-2024 The PHP Group](#)
- [My PHP.net](#)
- [Contact](#)
- [Other PHP.net sites](#)
- [Privacy policy](#)

