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

Iterables »
« Strings

Arrays ¶

An array in PHP is actually an ordered map. A map is a type that associates *values* to *keys*. This type is optimized for several different uses; it can be treated as an array, list (vector), hash table (an implementation of a map), dictionary, collection, stack, queue, and probably more. As array values can be other arrays, trees and multidimensional arrays are also possible.

Explanation of those data structures is beyond the scope of this manual, but at least one example is provided for each of them. For more information, look towards the considerable literature that exists about this broad topic.

Syntax ¶

Specifying with array() ¶

An array can be created using the array() language construct. It takes any number of comma-separated *key => value* pairs as arguments.

```
array(
    key  => value,
    key2 => value2,
    key3 => value3,
    ...
)
```

The comma after the last array element is optional and can be omitted. This is usually done for single-line arrays, i.e. *array(1, 2)* is preferred over *array(1, 2,)*. For multi-line arrays on the other hand the trailing comma is commonly used, as it allows easier addition of new elements at the end.

As of PHP 5.4 you can also use the short array syntax, which replaces *array()* with *[]*.

Example #1 A simple array

```php
<?php
$array = array(
        "foo" => "bar",
        "bar" => "foo",
);

// as of PHP 5.4
$array = [
        "foo" => "bar",
        "bar" => "foo",
];
?>
```

The key can either be an integer or a string. The value can be of any type.

Additionally the following key casts will occur:

- Strings containing valid decimal integers, unless the number is preceded by a *+* sign, will be cast to the integer type. E.g. the key *"8"* will actually be stored under *8*. On the other hand *"08"* will not be cast, as it isn't a valid decimal integer.
- Floats are also cast to integers, which means that the fractional part will be truncated. E.g. the key *8.7* will actually be stored under *8*.
- Bools are cast to integers, too, i.e. the key *true* will actually be stored under *1* and the key *false* under *0*.
- Null will be cast to the empty string, i.e. the key *null* will actually be stored under *""*.

- Arrays and objects *can not* be used as keys. Doing so will result in a warning: *Illegal offset type*.

If multiple elements in the array declaration use the same key, only the last one will be used as all others are overwritten.

Example #2 Type Casting and Overwriting example

```
<?php
$array = array(
        1    => "a",
        "1"  => "b",
        1.5  => "c",
        true => "d",
);
var_dump($array);
?>
```

The above example will output:

```
array(1) {
  [1]=>
  string(1) "d"
}
```

As all the keys in the above example are cast to *1*, the value will be overwritten on every new element and the last assigned value *"d"* is the only one left over.

PHP arrays can contain integer and string keys at the same time as PHP does not distinguish between indexed and associative arrays.

Example #3 Mixed integer and string keys

```
<?php
$array = array(
        "foo" => "bar",
        "bar" => "foo",
        100   => -100,
        -100  => 100,
);
var_dump($array);
?>
```

The above example will output:

```
array(4) {
  ["foo"]=>
  string(3) "bar"
  ["bar"]=>
  string(3) "foo"
  [100]=>
  int(-100)
  [-100]=>
  int(100)
}
```

The key is optional. If it is not specified, PHP will use the increment of the largest previously used integer key.

Example #4 Indexed arrays without key

```
<?php
$array = array("foo", "bar", "hello", "world");
var_dump($array);
?>
```

The above example will output:

```
array(4) {
  [0]=>
  string(3) "foo"
  [1]=>
  string(3) "bar"
  [2]=>
  string(5) "hello"
  [3]=>
  string(5) "world"
}
```

It is possible to specify the key only for some elements and leave it out for others:

Example #5 Keys not on all elements

```
<?php
$array = array(
                "a",
                "b",
        6 => "c",
                "d",
);
var_dump($array);
?>
```

The above example will output:

```
array(4) {
  [0]=>
  string(1) "a"
  [1]=>
  string(1) "b"
  [6]=>
  string(1) "c"
  [7]=>
  string(1) "d"
}
```

As you can see the last value *"d"* was assigned the key *7*. This is because the largest integer key before that was *6*.

Accessing array elements with square bracket syntax ¶

Array elements can be accessed using the *array[key]* syntax.

Example #6 Accessing array elements

```
<?php
$array = array(
        "foo" => "bar",
        42    => 24,
        "multi" => array(
                "dimensional" => array(
                        "array" => "foo"
                )
        )
);

var_dump($array["foo"]);
var_dump($array[42]);
var_dump($array["multi"]["dimensional"]["array"]);
?>
```

The above example will output:

```
string(3) "bar"
int(24)
string(3) "foo"
```

> **Note**:
>
> Both square brackets and curly braces can be used interchangeably for accessing array elements (e.g. *$array[42]* and *$array{42}* will both do the same thing in the example above).

As of PHP 5.4 it is possible to array dereference the result of a function or method call directly. Before it was only possible using a temporary variable.

As of PHP 5.5 it is possible to array dereference an array literal.

Example #7 Array dereferencing

```
<?php
function getArray() {
        return array(1, 2, 3);
}

// on PHP 5.4
$secondElement = getArray()[1];

// previously
$tmp = getArray();
$secondElement = $tmp[1];

// or
list(, $secondElement) = getArray();
?>
```

> **Note**:
>
> Attempting to access an array key which has not been defined is the same as accessing any other undefined variable: an E_NOTICE-level error message will be issued, and the result will be NULL.

> **Note**:
>
> Array dereferencing a scalar value which is not a string silently yields NULL, i.e. without issuing an error message.

Creating/modifying with square bracket syntax ¶

An existing array can be modified by explicitly setting values in it.

This is done by assigning values to the array, specifying the key in brackets. The key can also be omitted, resulting in an empty pair of brackets (*[]*).

```
$arr[key] = value;
$arr[] = value;
// key may be an integer or string
// value may be any value of any type
```

If *$arr* doesn't exist yet, it will be created, so this is also an alternative way to create an array. This practice is however discouraged because if *$arr* already contains some value (e.g. string from request variable) then this value will stay in the place and *[]* may actually stand for string access operator. It is always better to initialize a variable by a direct assignment.

> **Note**: As of PHP 7.1.0, applying the empty index operator on a string throws a fatal error. Formerly, the string was silently converted to an array.

To change a certain value, assign a new value to that element using its key. To remove a key/value pair, call the unset() function on it.

```
<?php
$arr = array(5 => 1, 12 => 2);
```

```
$arr[] = 56;        // This is the same as $arr[13] = 56;
                    // at this point of the script

$arr["x"] = 42; // This adds a new element to
                    // the array with key "x"

unset($arr[5]); // This removes the element from the array

unset($arr);        // This deletes the whole array
?>
```

Note:

As mentioned above, if no key is specified, the maximum of the existing integer indices is taken, and the new key will be that maximum value plus 1 (but at least 0). If no integer indices exist yet, the key will be *0* (zero).

Note that the maximum integer key used for this *need not currently exist in the array*. It need only have existed in the array at some time since the last time the array was re-indexed. The following example illustrates:

```
<?php
// Create a simple array.
$array = array(1, 2, 3, 4, 5);
print_r($array);

// Now delete every item, but leave the array itself intact:
foreach ($array as $i => $value) {
        unset($array[$i]);
}
print_r($array);

// Append an item (note that the new key is 5, instead of 0).
$array[] = 6;
print_r($array);

// Re-index:
$array = array_values($array);
$array[] = 7;
print_r($array);
?>
```

The above example will output:

```
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
    [3] => 4
    [4] => 5
)
Array
(
)
Array
(
    [5] => 6
)
Array
(
    [0] => 6
    [1] => 7
)
```

Useful functions ¶

There are quite a few useful functions for working with arrays. See the array functions section.

> **Note**:
>
> The unset() function allows removing keys from an array. Be aware that the array will *not* be reindexed. If a true "remove and shift" behavior is desired, the array can be reindexed using the array_values() function.

```
<?php
$a = array(1 => 'one', 2 => 'two', 3 => 'three');
unset($a[2]);
/* will produce an array that would have been defined as
      $a = array(1 => 'one', 3 => 'three');
      and NOT
      $a = array(1 => 'one', 2 =>'three');
*/

$b = array_values($a);
// Now $b is array(0 => 'one', 1 =>'three')
?>
```

The foreach control structure exists specifically for arrays. It provides an easy way to traverse an array.

Array do's and don'ts ¶

Why is *$foo[bar]* wrong? ¶

Always use quotes around a string literal array index. For example, *$foo['bar']* is correct, while *$foo[bar]* is not. But why? It is common to encounter this kind of syntax in old scripts:

```
<?php
$foo[bar] = 'enemy';
echo $foo[bar];
// etc
?>
```

This is wrong, but it works. The reason is that this code has an undefined constant (*bar*) rather than a string (*'bar'* - notice the quotes). It works because PHP automatically converts a *bare string* (an unquoted string which does not correspond to any known symbol) into a string which contains the bare string. For instance, if there is no defined constant named `bar`, then PHP will substitute in the string *'bar'* and use that.

> **Note**: This does not mean to *always* quote the key. Do not quote keys which are constants or variables, as this will prevent PHP from interpreting them.

```
<?php
error_reporting(E_ALL);
ini_set('display_errors', true);
ini_set('html_errors', false);
// Simple array:
$array = array(1, 2);
$count = count($array);
for ($i = 0; $i < $count; $i++) {
        echo "\nChecking $i: \n";
        echo "Bad: " . $array['$i'] . "\n";
        echo "Good: " . $array[$i] . "\n";
        echo "Bad: {$array['$i']}\n";
        echo "Good: {$array[$i]}\n";
}
?>
```

The above example will output:

```
Checking 0:
Notice: Undefined index:  $i in /path/to/script.html on line 9
Bad:
Good: 1
Notice: Undefined index:  $i in /path/to/script.html on line 11
Bad:
Good: 1

Checking 1:
Notice: Undefined index:  $i in /path/to/script.html on line 9
Bad:
Good: 2
Notice: Undefined index:  $i in /path/to/script.html on line 11
Bad:
Good: 2
```

More examples to demonstrate this behaviour:

```php
<?php
// Show all errors
error_reporting(E_ALL);

$arr = array('fruit' => 'apple', 'veggie' => 'carrot');

// Correct
print $arr['fruit'];   // apple
print $arr['veggie']; // carrot

// Incorrect.   This works but also throws a PHP error of level E_NOTICE because
// of an undefined constant named fruit
//
// Notice: Use of undefined constant fruit - assumed 'fruit' in...
print $arr[fruit];       // apple

// This defines a constant to demonstrate what's going on.   The value 'veggie'
// is assigned to a constant named fruit.
define('fruit', 'veggie');

// Notice the difference now
print $arr['fruit'];   // apple
print $arr[fruit];       // carrot

// The following is okay, as it's inside a string. Constants are not looked for
// within strings, so no E_NOTICE occurs here
print "Hello $arr[fruit]";           // Hello apple

// With one exception: braces surrounding arrays within strings allows constants
// to be interpreted
print "Hello {$arr[fruit]}";       // Hello carrot
print "Hello {$arr['fruit']}";   // Hello apple

// This will not work, and will result in a parse error, such as:
// Parse error: parse error, expecting T_STRING' or T_VARIABLE' or T_NUM_STRING'
// This of course applies to using superglobals in strings as well
print "Hello $arr['fruit']";
print "Hello $_GET['foo']";

// Concatenation is another option
print "Hello " . $arr['fruit']; // Hello apple
?>
```

When error_reporting is set to show E_NOTICE level errors (by setting it to E_ALL, for example), such uses will become immediately visible. By default, error_reporting is set not to show

notices.

As stated in the syntax section, what's inside the square brackets ('*[*' and '*]*') must be an expression. This means that code like this works:

```
<?php
echo $arr[somefunc($bar)];
?>
```

This is an example of using a function return value as the array index. PHP also knows about constants:

```
<?php
$error_descriptions[E_ERROR]     = "A fatal error has occurred";
$error_descriptions[E_WARNING]  = "PHP issued a warning";
$error_descriptions[E_NOTICE]    = "This is just an informal notice";
?>
```

Note that `E_ERROR` is also a valid identifier, just like *bar* in the first example. But the last example is in fact the same as writing:

```
<?php
$error_descriptions[1] = "A fatal error has occurred";
$error_descriptions[2] = "PHP issued a warning";
$error_descriptions[8] = "This is just an informal notice";
?>
```

because `E_ERROR` equals *1*, etc.

So why is it bad then?

At some point in the future, the PHP team might want to add another constant or keyword, or a constant in other code may interfere. For example, it is already wrong to use the words *empty* and *default* this way, since they are reserved keywords.

> **Note**: To reiterate, inside a double-quoted string, it's valid to not surround array indexes with quotes so *"$foo[bar]"* is valid. See the above examples for details on why as well as the section on variable parsing in strings.

Converting to array ¶

For any of the types integer, float, string, boolean and resource, converting a value to an array results in an array with a single element with index zero and the value of the scalar which was converted. In other words, *(array)$scalarValue* is exactly the same as *array($scalarValue)*.

If an object is converted to an array, the result is an array whose elements are the object's properties. The keys are the member variable names, with a few notable exceptions: integer properties are unaccessible; private variables have the class name prepended to the variable name; protected variables have a '*' prepended to the variable name. These prepended values have null bytes on either side. This can result in some unexpected behaviour:

```
<?php

class A {
    private $A; // This will become '\0A\0A'
}

class B extends A {
    private $A; // This will become '\0B\0A'
    public $AA; // This will become 'AA'
}
```

```
var_dump((array) new B());
?>
```

The above will appear to have two keys named 'AA', although one of them is actually named '\0A\0A'.

Converting NULL to an array results in an empty array.

Comparing ¶

It is possible to compare arrays with the array_diff() function and with array operators.

Examples ¶

The array type in PHP is very versatile. Here are some examples:

```
<?php
// This:
$a = array( 'color' => 'red',
            'taste' => 'sweet',
            'shape' => 'round',
            'name' => 'apple',
            4 // key will be 0
          );

$b = array('a', 'b', 'c');

// . . .is completely equivalent with this:
$a = array();
$a['color'] = 'red';
$a['taste'] = 'sweet';
$a['shape'] = 'round';
$a['name'] = 'apple';
$a[] = 4; // key will be 0

$b = array();
$b[] = 'a';
$b[] = 'b';
$b[] = 'c';

// After the above code is executed, $a will be the array
// array('color' => 'red', 'taste' => 'sweet', 'shape' => 'round',
// 'name' => 'apple', 0 => 4), and $b will be the array
// array(0 => 'a', 1 => 'b', 2 => 'c'), or simply array('a', 'b', 'c').
?>
```

Example #8 Using array()

```
<?php
// Array as (property-)map
$map = array( 'version' => 4,
              'OS' => 'Linux',
              'lang' => 'english',
              'short_tags' => true
            );

// strictly numerical keys
$array = array( 7,
                8,
                0,
                156,
```

```
                                -10
                            );
// this is the same as array(0 => 7, 1 => 8, ...)

$switching = array(                 10, // key = 0
                                     5       =>    6,
                                     3       =>    7,
                                     'a'    =>    4,
                                                    11, // key = 6 (maximum of integer-
indices was 5)
                                     '8'    =>    2, // key = 8 (integer!)
                                     '02'  => 77, // key = '02'
                                     0       => 12    // the value 10 will be overwritten by 12
                                 );

// empty array
$empty = array();
?>
```

Example #9 Collection

```
<?php
$colors = array('red', 'blue', 'green', 'yellow');

foreach ($colors as $color) {
        echo "Do you like $color?\n";
}

?>
```

The above example will output:

```
Do you like red?
Do you like blue?
Do you like green?
Do you like yellow?
```

Changing the values of the array directly is possible by passing them by reference.

Example #10 Changing element in the loop

```
<?php
foreach ($colors as &$color) {
        $color = strtoupper($color);
}
unset($color); /* ensure that following writes to
$color will not modify the last array element */

print_r($colors);
?>
```

The above example will output:

```
Array
(
    [0] => RED
    [1] => BLUE
    [2] => GREEN
    [3] => YELLOW
)
```

This example creates a one-based array.

Example #11 One-based index

```
<?php
$firstquarter  = array(1 => 'January', 'February', 'March');
print_r($firstquarter);
?>
```

The above example will output:

```
Array
(
    [1] => 'January'
    [2] => 'February'
    [3] => 'March'
)
```

Example #12 Filling an array

```
<?php
// fill an array with all items from a directory
$handle = opendir('.');
while (false !== ($file = readdir($handle))) {
        $files[] = $file;
}
closedir($handle);
?>
```

Arrays are ordered. The order can be changed using various sorting functions. See the array functions section for more information. The count() function can be used to count the number of items in an array.

Example #13 Sorting an array

```
<?php
sort($files);
print_r($files);
?>
```

Because the value of an array can be anything, it can also be another array. This enables the creation of recursive and multi-dimensional arrays.

Example #14 Recursive and multi-dimensional arrays

```
<?php
$fruits = array ( "fruits"  => array ( "a" => "orange",
                                       "b" => "banana",
                                       "c" => "apple"
                                     ),
                  "numbers" => array ( 1,
                                       2,
                                       3,
                                       4,
                                       5,
                                       6
                                     ),
                  "holes"   => array ( "first",
                                       5 => "second",
                                            "third"
                                     )
                );

// Some examples to address values in the array above
echo $fruits["holes"][5];    // prints "second"
echo $fruits["fruits"]["a"]; // prints "orange"
unset($fruits["holes"][0]);  // remove "first"
```

```
// Create a new multi-dimensional array
$juices["apple"]["green"] = "good";
?>
```

Array assignment always involves value copying. Use the reference operator to copy an array by reference.

```
<?php
$arr1 = array(2, 3);
$arr2 = $arr1;
$arr2[] = 4; // $arr2 is changed,
                        // $arr1 is still array(2, 3)

$arr3 = &$arr1;
$arr3[] = 4; // now $arr1 and $arr3 are the same
?>
```

add a note

User Contributed Notes 21 notes

up
down
82
mlvljr ¶
6 years ago
please note that when arrays are copied, the "reference status" of their members is preserved (http://www.php.net/manual/en/language.references.whatdo.php).

up
down
24
thomas tulinsky ¶
1 year ago
I think your first, main example is needlessly confusing, very confusing to newbies:

```
$array = array(
      "foo" => "bar",
      "bar" => "foo",
);
```

It should be removed.

For newbies:
An array index can be any string value, even a value that is also a value in the array.
The value of array["foo"] is "bar".
The value of array["bar"] is "foo"

The following expressions are both true:
$array["foo"] == "bar"
$array["bar"] == "foo"

up
down
48
ken underscore yap atsign email dot com ¶
9 years ago
"If you convert a NULL value to an array, you get an empty array."

This turns out to be a useful property. Say you have a search function that returns an array of values on success or NULL if nothing found.

```
<?php $values = search(...); ?>
```

Now you want to merge the array with another array. What do we do if $values is NULL? No problem:

```
<?php $combined = array_merge((array)$values, $other); ?>
```

Voila.

up
down
8
as at asgl dot de ¶
8 months ago

```
// On PHP 7.1 until Jan. 20 2017
$testVar = "";
$testVar[2] = "Meine eigene Lösung";
echo $testVar[2];

// Result:
// Meine eigene Lösung => $testVar is an ARRAY

// On PHP 7.1.1 after Jan. 20 2017
$testVar = "";
$testVar[2] = "Meine eigene Lösung";
echo $testVar[2];
// Result:
// M => $testVar is a STRING !!!
```

up
down
45
jeff splat codedread splot com ¶
12 years ago

Beware that if you're using strings as indices in the $_POST array, that periods are transformed into underscores:

```
<html>
<body>
<?php
      printf("POST: "); print_r($_POST); printf("<br/>");
?>
<form method="post" action="<?php echo $_SERVER['PHP_SELF']; ?>">
      <input type="hidden" name="Windows3.1" value="Sux">
      <input type="submit" value="Click" />
</form>
</body>
</html>
```

Once you click on the button, the page displays the following:

```
POST: Array ( [Windows3_1] => Sux )
```

up
down
29
chris at ocportal dot com ¶
4 years ago

Note that array value buckets are reference-safe, even through serialization.

```
<?php
$x='initial';
$test=array('A'=>&$x,'B'=>&$x);
$test=unserialize(serialize($test));
$test['A']='changed';
```

```
echo $test['B']; // Outputs "changed"
?>
```

This can be useful in some cases, for example saving RAM within complex structures.

up
down
35
lars-phpcomments at ukmix dot net ¶
12 years ago

Used to creating arrays like this in Perl?

```
@array = ("All", "A".."Z");
```

Looks like we need the range() function in PHP:

```
<?php
$array = array_merge(array('All'), range('A', 'Z'));
?>
```

You don't need to array_merge if it's just one range:

```
<?php
$array = range('A', 'Z');
?>
```

up
down
19
ivegner at yandex dot ru ¶
4 years ago

Note that objects of classes extending ArrayObject SPL class are treated as arrays, and not as objects when converting to array.

```
<?php
class ArrayObjectExtended extends ArrayObject
{
      private $private = 'private';
      public $hello = 'world';
}

$object = new ArrayObjectExtended();
$array = (array) $object;

// This will not expose $private and $hello properties of $object,
// but return an empty array instead.
var_export($array);
?>
```

up
down
12
mathiasgrimm at gmail dot com ¶
3 years ago

```
<?php

$a['a'] = null;
$a['b'] = array();

echo $a['a']['non-existent']; // DOES NOT throw an E_NOTICE error as expected.

echo $a['b']['non-existent']; // throws an E_NOTICE as expected
?>
```

I added this bug to bugs.php.net (https://bugs.php.net/bug.php?id=68110)
however I made tests with php4, 5.4 and 5.5 versions and all behave the same way.

This, in my point of view, should be cast to an array type and throw the same error.

This is, according to the documentation on this page, wrong.

From doc:
"Note:
Attempting to access an array key which has not been defined is the same as accessing any other undefined variable: an E_NOTICE-level error message will be issued, and the result will be NULL."

up
down
28

ia [AT] zoznam [DOT] sk ¶

12 years ago

Regarding the previous comment, beware of the fact that reference to the last value of the array remains stored in $value after the foreach:

```
<?php
foreach ( $arr as $key => &$value )
{
      $value = 1;
}

// without next line you can get bad results...
//unset( $value );

$value = 159;
?>
```

Now the last element of $arr has the value of '159'. If we remove the comment in the unset() line, everything works as expected ($arr has all values of '1').

Bad results can also appear in nested foreach loops (the same reason as above).

So either unset $value after each foreach or better use the longer form:

```
<?php
foreach ( $arr as $key => $value )
{
      $arr[ $key ] = 1;
}
?>
```

up
down
5

note dot php dot lorriman at spamgourmet dot org ¶

3 years ago

There is another kind of array (php>= 5.3.0) produced by

```
$array = new SplFixedArray(5);
```

Standard arrays, as documented here, are marvellously flexible and, due to the underlying hashtable, extremely fast for certain kinds of lookup operation.

Supposing a large string-keyed array

```
$arr=['string1'=>$data1, 'string2'=>$data2 etc....]
```

when getting the keyed data with

$data=$arr['string1'];

php does *not* have to search through the array comparing each key string to the given key ('string1') one by one, which could take a long time with a large array. Instead the hashtable means that php takes the given key string and computes from it the memory location of the keyed data, and then instantly retrieves the data. Marvellous! And so quick. And no need to know anything about hashtables as it's all hidden away.

However, there is a lot of overhead in that. It uses lots of memory, as hashtables tend to (also nearly doubling on a 64bit server), and should be significantly slower for integer keyed arrays than old-fashioned (non-hashtable) integer-keyed arrays. For that see more on SplFixedArray :

http://uk3.php.net/SplFixedArray

Unlike a standard php (hashtabled) array, if you lookup by integer then the integer itself denotes the memory location of the data, no hashtable computation on the integer key needed. This is much quicker. It's also quicker to build the array compared to the complex operations needed for hashtables. And it uses a lot less memory as there is no hashtable data structure. This is really an optimisation decision, but in some cases of large integer keyed arrays it may significantly reduce server memory and increase performance (including the avoiding of expensive memory deallocation of hashtable arrays at the exiting of the script).

up
down
14
caifara aaaat im dooaat be ¶
12 years ago

[Editor's note: You can achieve what you're looking for by referencing $single, rather than copying it by value in your foreach statement. See http://php.net/foreach for more details.]

Don't know if this is known or not, but it did eat some of my time and maybe it won't eat your time now...

I tried to add something to a multidimensional array, but that didn't work at first, look at the code below to see what I mean:

```
<?php

$a1 = array( "a" => 0, "b" => 1 );
$a2 = array( "aa" => 00, "bb" => 11 );

$together = array( $a1, $a2 );

foreach( $together as $single ) {
      $single[ "c" ] = 3 ;
}

print_r( $together );
/* nothing changed result is:
Array
(
      [0] => Array
            (
                  [a] => 0
                  [b] => 1
            )
      [1] => Array
            (
                  [aa] => 0
                  [bb] => 11
            )
```

```
) */

foreach( $together as $key => $value ) {
      $together[$key]["c"] = 3 ;
}

print_r( $together );

/* now it works, this prints
Array
(
      [0] => Array
            (
                  [a] => 0
                  [b] => 1
                  [c] => 3
            )
      [1] => Array
            (
                  [aa] => 0
                  [bb] => 11
                  [c] => 3
            )
)
*/

?>
```

up
down
1
Walter Tross ¶
7 years ago

It is true that "array assignment always involves value copying", but the copy is a "lazy copy". This means that the data of the two variables occupy the same memory as long as no array element changes.

E.g., if you have to pass an array to a function that only needs to read it, there is no advantage at all in passing it by reference.

up
down
-1
Yesterday php'er ¶
8 months ago

--- quote ---
Note:
Both square brackets and curly braces can be used interchangeably for accessing array elements
--- quote end ---

At least for php 5.4 and 5.6; if function returns an array, the curly brackets does not work directly accessing function result, eg. WillReturnArray(){1} . This will give "syntax error, unexpected '{' in...".
Personally I use only square brackets, expect for accessing single char in string. Old habits...

up
down
-5
martijntje at martijnotto dot nl ¶
5 years ago

Please note that adding the magic __toString() method to your objects will not allow you to seek an array with it, it still throws an Illegal Offset warning.

The solution is to cast it to a string first, like this

```
$array[(string) $stringableObject]
```

up
down
-6
brta dot akos at gmail dot com ¶
3 years ago

```
Why not to user one-based arrays:

<?php
$a   = array(1 => 'a', 'b', 'd');
print_r($a);
array_splice($a,2,0,'c');
print_r($a);
?>

output:
Array ( [1] => a [2] => b [3] => d ) Array ( [0] => a [1] => b [2] => c [3] => d )
```

up
down
-8
aditycse at gmail dot com ¶
2 years ago

```
/*
* Name : Aditya Mehrotra
* Email: aditycse@gmail.com
*/
<?php
//Array can have following data type in key i.e string,integer
//Behaviour of array in case of array key has data type float or double

$exampleArray = array(0,
      1,
      "2.99999999" => 56,
      2 => 2,
      3.9999 => 3,
      3 => 3.1,
      true => 4,
      false => 6,
);

//array structure
print_r($exampleArray);
/* Array
   (
         [0] => 6
         [1] => 4
         [2.99999999] => 56
         [2] => 2
         [3] => 3.1
   )
*/

//array of array keys
print_r(array_keys($exampleArray));
/*
Array
(
      [0] => 0
      [1] => 1
```

```
    [2] => 2.99999999
    [3] => 2
    [4] => 3
)
*/
```

up
down
-10

php at markuszeller dot com ¶

1 year ago

Sometimes I need to match fieldnames from database tables. But if a source field is used many times you can not use a hash "=>", because it overrides the key.

My approach is to use a comma separated array and use a while-loop in conjunction with each. Having that you can iterate key/value based, but may have a key multiple times.

```
$fieldmap = array
(
    'id', 'import_id',
    'productname', 'title',
    'datetime_online', 'onlineDate',
    'datetime_test_final', 'offlineDate',
    'active', 'status',
    'questionaire_intro', 'text_lead',
    'datetime_online', 'createdAt',
    'datetime_online', 'updatedAt'
);

while(list(,$key) = each($fieldmap))
{
    list(,$value) = each($fieldmap);
    echo "$key: $value\n";
}
```

up
down
-20

Anonymous ¶

11 years ago

This page should include details about how associative arrays are implemened inside PHP; e.g. using hash-maps or b-trees.

This has important implictions on the permance characteristics of associative arrays and how they should be used; e.g. b-tree are slow to insert but handle collisions better than hashmaps. Hashmaps are faster if there are no collisions, but are slower to retrieve when there are collisions. These factors have implictions on how associative arrays should be used.

up
down
-22

Spudley ¶

10 years ago

On array recursion...

Given the following code:

```
<?php
$myarray = array('test',123);
$myarray[] = &$myarray;
print_r($myarray);
?>
```

The print_r() will display *RECURSION* when it gets to the third element of the array.

There doesn't appear to be any other way to scan an array for recursive references, so if you need to check for them, you'll have to use print_r() with its second parameter to capture the output and look for the word *RECURSION*.

It's not an elegant solution, but it's the only one I've found, so I hope it helps someone.

up
down
-84
carl at linkleaf dot com ¶
10 years ago

Its worth noting that there does not appear to be any functional limitations on the length or content of string indexes. The string indexes for your arrays can contain any characters, including new line characters, and can be of any length:

```
<?php

$key = "XXXXX";
$test = array($key => "test5");

for ($x = 0; $x < 500; $x++) {
   $key .= "X";
   $value = "test" . strlen($key);
   $test[$key] = $value;
}

echo "<pre>";
print_r($test);
echo "</pre>";

?>
```

Keep in mind that using extremely long array indexes is not a good practice and could cost you lots of extra CPU time. However, if you have to use a long string as an array index you won't have to worry about the length or content.

add a note