Search

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

Операторы »
« Магические константы

- Руководство по PHP
- Справочник языка

Change language: Russian

Submit a Pull Request Report a Bug

Выражения

Выражения — это самые важные строительные элементы PHP. Почти всё, что разработчик пишет в PHP, — выражение. Самое простое и точное определение выражения — «всё, у чего есть значение».

Основные формы выражений — это константы и переменные. Если записать `$a = 5`, значение `5` будет присвоено переменной *$a*. У значения `5`, очевидно, — значение 5 или, говоря по-другому, `5` — это выражение со значением 5 (в данном случае `5` — это целочисленная константа).

После этого присвоения ожидается, что значение переменной *$a* тоже равно 5, поэтому, если написано `$b = $a`, ожидается, что работать это будет так же, как если бы было написано `$b = 5`. Говоря по-другому, переменная *$a* — это также выражение со значением 5. Если всё работает верно, то так и произойдёт.

Немного более сложные примеры выражений — это функции. Например, рассмотрим следующую функцию:

```
<?php

function foo ()
{
return 5;
}
?>
```

Опираясь на то, что разработчик знаком принципом работы функций (если нет, рекомендовано прочитать главу о функциях), он может предположить, что напечатать `$c = foo()` — по смыслу то же, что написать `$c = 5`, и будет прав. Функции — это выражения, значение которых — то, что возвращает функция. Поскольку функция `foo()` возвращает 5, значением выражения «`foo()`» будет 5. Обычно функции не просто возвращают статическое значение, а что-то вычисляют.

Конечно, значения в PHP не обязаны быть целочисленными, и очень часто это не так. Язык PHP поддерживает четыре типа скалярных значений: целочисленные значения (int), значения с плавающей точкой (float), строковые (string) и логические (bool) значения (скалярные значения это те, которые нельзя «разбить» на меньшие части, в отличие, например, от массивов). PHP поддерживает также два сложных (составных, композитных — нескалярных) типа: массивы и объекты. Каждое из этих типов значений разрешено присваивать переменной или возвращать функцией.

Однако PHP использует выражения значительно шире, точно так же, как это делают другие языки. PHP — это язык с ориентацией на выражения, который почти всё в коде рассматривает как выражение. Вернёмся к тому примеру, с которым мы уже имели дело: `$a = 5`. Легко заметить, что здесь есть два значения — значение целочисленной константы `5` и значение переменной *$a*, также принимающей значение 5. Но, хотя это неочевидно, здесь есть ещё одно значение — значение самого присвоения. Само присвоение вычисляется в присвоенное значение, в данном случае — в 5. То есть выражение `$a = 5`, независимо от того, что оно делает, — это выражение со значением 5. Поэтому запись `$b = ($a = 5)` равносильна записи `$a = 5; $b = 5;` (точка с запятой означает конец выражения). Поскольку операции присвоения анализируются справа налево, также разрешено написать `$b = $a = 5`.

Другой хороший пример ориентированности на выражения — префиксный и постфиксный инкремент и декремент. Пользователи PHP и других языков возможно уже знакомы с формой записи `variable++` и `variable--`. Это операторы инкремента и декремента. Как и язык Си, язык PHP поддерживает два типа инкремента — префиксный и постфиксный. Они оба инкрементируют значение переменной и эффект их действия на неё одинаков. Разница состоит в значении выражения инкремента. Префиксный инкремент, записываемый как `++$variable`, вычисляется в инкрементированное значение (PHP инкрементирует переменную до того как прочесть её значение, отсюда название «преинкремент»). Постфиксный инкремент, записываемый как `$variable++`, вычисляется в первоначальное значение переменной *$variable* до её приращения (PHP вначале читает значение переменной и только потом инкрементирует её, отсюда название «постинкремент»).

Распространённые типы выражений — это выражения сравнения. Эти выражения оцениваются либо как **`false`** (ложь), либо как **`true`** (истина). PHP поддерживает операции сравнения > (больше), >= (больше либо равно), == (равно), != (не равно), < (меньше) и <= (меньше либо равно). Он также поддерживает операторы строгого равенства: === (равно и одного типа) и !== (не равно или не одного типа). Обычно этими выражениями пользуются в операторах условного выполнения, например, `if`.

Последний пример выражений, который будет здесь рассмотрен, это смешанные выражения операции и присвоения. Мы уже знаем, что если нужно увеличить значение переменной *$a* на 1, можно просто написать `$a++` или `++$a`. Но что, если нужно прибавить больше единицы, например, 3? Можно было бы написать `$a++` много раз, однако, очевидно, это не очень рационально и удобно. Более распространённая практика — запись вида `$a = $a + 3`. Выражение `$a + 3`

вычисляется в значение переменной *$a*, к которому прибавлено 3 и снова присвоено значение переменной *$a*, увеличивая в результате значение переменной *$a* на 3. В PHP, как и в ряде других языков, например Си, разрешено записать это более коротким образом, что увеличит очевидность смысла и скорость понимания кода по прошествии времени. Прибавить 3 к текущему значению переменной *$a* можно, записав `$a += 3`. Это означает дословно «взять значение *$a*, прибавить к нему 3 и снова присвоить его переменной *$a*». Кроме большей понятности и краткости, это быстрее работает. Значением выражения `$a += 3`, как и обычного присвоения, будет присвоенное значение. Обратите внимание, что это НЕ 3, а суммированное значение переменной *$a* плюс 3 (то, что было присвоено перменной *$a*). Поэтому разрешено использовать любой бинарный оператор, например, `$a -= 5` (вычесть 5 из значения переменной *$a*), `$b *= 7` (умножить значение переменной *$b* на 7) и т. д.

Существует ещё одно выражение, которое может выглядеть необычно, если вы не встречали его в других языках — тернарный условный оператор:

```
<?php

$first ? $second : $third
?>
```

Если значение первого подвыражения — **`true`** (ненулевое значение), то выполняется второе подвыражение, которое и будет результатом условного выражения. Или выполняется третье подвыражение и его значение будет результатом.

Задача следующего примера — помочь немного улучшить понимание префиксного и постфиксного инкремента и выражений:

```
<?php

function double($i)
{
return $i*2;
}

$b = $a = 5; /* присвоить значение пять переменным $a и $b */
$c = $a++; /* постфиксный инкремент, присвоить значение переменной
$a (5) — переменной $c */
$e = $d = ++$b; /* префиксный инкремент, присвоить увеличенное
значение переменной $b (6) — переменным $d и $e */

/* в этой точке и переменная $d, и переменная $e равны 6 */

$f = double($d++); /* присвоить удвоенное значение перменной $d перед
инкрементом (2 * 6 = 12) — переменной $f */
$g = double(++$e); /* присвоить удвоенное значение перменной $e после
инкремента (2 * 7 = 14) — переменной $g */
$h = $g += 10; /* сначала переменная $g увеличивается на 10,
приобретая, в итоге, значение 24. Затем значение
присвоения (24) присваивается переменной $h,
которая в итоге также становится равной 24. */
?>
```

Иногда выражения рассматриваются как инструкции. В данном случае у инструкции следующий вид — «`expr ;`» — выражение со следующей за ним точкой с запятой. В записи `$b = $a = 5;`, `$a = 5` — это верное выражение, но оно само — не инструкция. Тогда как выражение `$b = $a = 5;` — верная инструкция.

Последнее, что нужно вспомнить, — это истинность значения выражений. Обычно в условных операторах и циклах может интересовать не конкретное значение выражения, а только его истинность (значение **`true`** или **`false`**). Константы **`true`** и **`false`** (регистронезависимые) — это два возможных логических значения. Выражения разрешено автоматически преобразовать в логическое значение. Подробнее о том, как это сделать, рассказано в разделе о приведении типов.

Язык PHP со всей полнотой и мощностью реализует выражения, и их полное документирование выходит за рамки этого руководства. Примеры выше дают представление о выражениях, о том, что они из себя представляют, и как можно создавать полезные выражения. Для обозначения любого верного выражения PHP в этой документации будет использовано сокращение *expr*.

User Contributed Notes 12 notes

up
down
55
Magnus Deininger, dma05 at web dot de ¶
14 years ago

Note that even though PHP borrows large portions of its syntax from C, the ',' is treated quite differently. It's not possible to create combined expressions in PHP using the comma-operator that C has, except in for() loops.

Example (parse error):

```
<?php

$a = 2, $b = 4;

echo $a."\n";
echo $b."\n";

?>
```

Example (works):

```
<?php

for ($a = 2, $b = 4; $a < 3; $a++)
{
echo $a."\n";
echo $b."\n";
}

?>
```

This is because PHP doesn't actually have a proper comma-operator, it's only supported as syntactic sugar in for() loop headers. In C, it would have been perfectly legitimate to have this:

```
int f()
{
int a, b;
a = 2, b = 4;

return a;
}
```

or even this:

```
int g()
{
int a, b;
a = (2, b = 4);

return a;
}
```

In f(), a would have been set to 2, and b would have been set to 4.
In g(), (2, b = 4) would be a single expression which evaluates to 4, so both a and b would have been set to 4.

up
down
47
yasuo_ohgaki at hotmail dot com ¶
22 years ago

Manual defines "expression is anything that has value", Therefore, parser will give error for following code.

```
<?php
($val) ? echo('true') : echo('false');
Note: "? : " operator has this syntax "expr ? expr : expr;"
?>
```

since echo does not have(return) value and ?: expects expression(value).

However, if function/language constructs that have/return value, such as include(), parser compiles code.

Note: User defined functions always have/return value without explicit return statement (returns NULL if there is no return statement). Therefore, user defined functions are always valid expressions.
[It may be useful to have VOID as new type to prevent programmer to use function as RVALUE by mistake]

For example,

```
<?php
($val) ? include('true.inc') : include('false.inc');
?>
```

is valid, since "include" returns value.

The fact "echo" does not return value(="echo" is not a expression), is less obvious to me.

Print() and Echo() is NOT identical since print() has/returns value and can be a valid expression.

up
down
20

chriswarbo at gmail dot com ¶

10 years ago

Note that there is a difference between a function and a function call, and both are expressions. PHP has two kinds of function, "named functions" and "anonymous functions". Here's an example with both:

```
<?php
// A named function. Its name is "double".
function double($x) {
return 2 * $x;
}

// An anonymous function. It has no name, in the same way that the string
// "hello" has no name. Since it is an expression, we can give it a temporary
// name by assigning it to the variable $triple.
$triple = function($x) {
return 3 * $x;
};
?>
```

We can "call" (or "run") both kinds of function. A "function call" is an expression with the value of whatever the function returns. For example:

```
<?php
// The easiest way to run a function is to put () after its name, containing its
// arguments (if any)
$my_numbers = array(double(5), $triple(5));
?>
```

$my_numbers is now an array containing 10 and 15, which are the return values of double and $triple when applied to the number 5.

Importantly, if we *don't* call a function, ie. we don't put () after its name,

then we still get expressions. For example:

```
<?php
$my_functions = array('double', $triple);
?>
```

$my_functions is now an array containing these two functions. Notice that named functions are more awkward than anonymous functions. PHP treats them differently because it didn't use to have anonymous functions, and the way named functions were implemented didn't work for anonymous functions when they were eventually added.

This means that instead of using a named function literally, like we can with anonymous functions, we have to use a string containing its name instead. PHP makes sure that these strings will be treated as functions when it's appropriate. For example:

```
<?php
$temp = 'double';
$my_number = $temp(5);
?>
```

$my_number will be 10, since PHP has spotted that we're treating a string as if it were a function, so it has looked up that named function for us.

Unfortunately PHP's parser is very quirky; rather than looking for generic patterns like "x(y)" and seeing if "x" is a function, it has lots of special-cases like "$x(y)". This makes code like "'double'(5)" invalid, so we have to do tricks like using temporary variables. There is another way around this restriction though, and that is to pass our functions to the "call_user_func" or "call_user_func_array" functions when we want to call them. For example:

```
<?php
$my_numbers = array(call_user_func('double', 5), call_user_func($triple, 5));
?>
```

$my_numbers contains 10 and 15 because "call_user_func" called our functions for us. This is possible because the string 'double' and the anonymous function $triple are expressions. Note that we can even use this technique to call an anonymous function without ever giving it a name:

```
<?php
$my_number = call_user_func(function($x) { return 4 * $x; }, 5);
?>
```

$my_number is now 20, since "call_user_func" called the anonymous function, which quadruples its argument, with the value 5.

Passing functions around as expressions like this is very useful whenever we need to use a 'callback'. Great examples of this are array_map and array_reduce.

19
Mattias at mail dot ee ¶
21 years ago

A note about the short-circuit behaviour of the boolean operators.

1. if (func1() || func2())
Now, if func1() returns true, func2() isn't run, since the expression will be true anyway.

```
2. if (func1() && func2())
Now, if func1() returns false, func2() isn't run, since the expression
will be false anyway.

The reason for this behaviour comes probably from the programming
language C, on which PHP seems to be based on. There the
short-circuiting can be a very useful tool. For example:

int * myarray = a_func_to_set_myarray(); // init the array
if (myarray != NULL && myarray[0] != 4321) // check
myarray[0] = 1234;

Now, the pointer myarray is checked for being not null, then the
contents of the array is validated. This is important, because if
you try to access an array whose address is invalid, the program
will crash and die a horrible death. But thanks to the short
circuiting, if myarray == NULL then myarray[0] won't be accessed,
and the program will work fine.
```

up
down
13

egonfreeman at gmail dot com ¶

16 years ago

```
It is worthy to mention that:

$n = 3;
$n * --$n

WILL RETURN 4 instead of 6.

It can be a hard to spot "error", because in our human thought process this really isn't an error at all! But you have to
remember that PHP (as it is with many other high-level languages) evaluates its statements RIGHT-TO-LEFT, and therefore "-
-$n" comes BEFORE multiplying, so - in the end - it's really "2 * 2", not "3 * 2".

It is also worthy to mention that the same behavior will change:

$n = 3;
$n * $n++

from 3 * 3 into 3 * 4. Post- operations operate on a variable after it has been 'checked', but it doesn't necessarily
state that it should happen AFTER an evaluation is over (on the contrary, as a matter of fact).

So, if you ever find yourself on a 'wild goose chase' for a bug in that "impossible-to-break, so-very-simple" piece of
code that uses pre-/post-'s, remember this post. :)

(just thought I'd check it out - turns out I was right :P)
```

up
down
15

winks716 ¶

16 years ago

```
reply to egonfreeman at gmail dot com
04-Apr-2007 07:45

the second example u mentioned as follow:
====================================

$n = 3;
$n * $n++

from 3 * 3 into 3 * 4. Post- operations operate on a variable after it has been 'checked', but it doesn't necessarily
state that it should happen AFTER an evaluation is over (on the contrary, as a matter of fact).
```

==========

everything works correctly but one sentence should be modified:

"from 3 * 3 into 3 * 4" should be "from 3 * 3 into 4 * 3"

best regards~ :)

up

down

9

petruzanauticoyahoo?com!ar ¶

16 years ago

Regarding the ternary operator, I would rather say that the best option is to enclose all the expression in parantheses, to avoid errors and improve clarity:

```
<?php
print ( $a > 1 ? "many" : "just one" );
?>
```

PS: for php, C++, and any other language that has it.

up

down

7

oliver at hankeln-online dot de ¶

21 years ago

The short-circuiting IS a feature. It is also available in C, so I suppose the developers won?t remove it in future PHP versions.

It is rather nice to write:

```
$file=fopen("foo","r") or die("Error!");
```

Greets,
Oliver

up

down

6

denzoo at gmail dot com ¶

15 years ago

To jvm at jvmyers dot com:
Your first two if statements just check if there's anything in the string, if you wish to actually execute the code in your string you need eval().

up

down

6

shawnster ¶

16 years ago

An easy fix (although intuitively tough to do...) is to reverse the comparison.

```
if (5 == $a) {}
```

If you forget the second '=', you'll get a parse error for trying to assign a value to a non-variable.

up

down

2

Bichis Paul ¶

7 years ago

Regarding 12345alex at gmx dot net's example:

I think you miss the identical equal documentation line from: http://php.net/manual/en/language.operators.comparison.php

```
$a == $b Equal TRUE if $a is equal to $b after type juggling.
$a === $b Identical TRUE if $a is equal to $b, and they are of the same type.

Try:
print array() === NULL ? "True" : "False";

Check this:
var_dump(is_null(array()));
```

up
down
2

antickon at gmail dot com ¶
11 years ago

```
evaluation order of subexpressions is not strictly defined for all operators

<?php
function a() {echo 'a';}
function b() {echo 'b';}
a() == b(); // outputs "ab", ie evaluates left-to-right

$a = 3;
var_dump( $a == $a = 4 ); // outputs bool(true), ie evaluates right-to-left
?>

this is not a bug: "we [php developers] make no guarantee about the order of evaluation".
See https://bugs.php.net/bug.php?id=61188
```

＋add a note