Expressions

Expressions are the most important building blocks of PHP. In PHP, almost anything you write is an expression. The simplest yet most accurate way to define an expression is "anything that has a value".

The most basic forms of expressions are constants and variables. When you type `$a = 5`, you're assigning `5` into `$a`. `5`, obviously, has the value 5, or in other words `5` is an expression with the value of 5 (in this case, `5` is an integer constant).

After this assignment, you'd expect `$a`'s value to be 5 as well, so if you wrote `$b = $a`, you'd expect it to behave just as if you wrote `$b = 5`. In other words, `$a` is an expression with the value of 5 as well. If everything works right, this is exactly what will happen.

Slightly more complex examples for expressions are functions. For instance, consider the following function:

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

Assuming you're familiar with the concept of functions (if you're not, take a look at the chapter about functions), you'd assume that typing `$c = foo()` is essentially just like writing `$c = 5`, and you're right. Functions are expressions with the value of their return value. Since `foo()` returns 5, the value of the expression '`foo()`' is 5. Usually functions don't just return a static value but compute something.

Of course, values in PHP don't have to be integers, and very often they aren't. PHP supports four scalar value types: `int` values, floating point values (`float`), `string` values and `bool` values (scalar values are values that you can't 'break' into smaller pieces, unlike arrays, for instance). PHP also supports two composite (non-scalar) types: arrays and objects. Each of these value types can be assigned into variables or returned from functions.

PHP takes expressions much further, in the same way many other languages do. PHP is an expression-oriented language, in the sense that almost everything is an expression. Consider the example we've already dealt with, `$a = 5`. It's easy to see that there are two values involved here,

the value of the integer constant `5`, and the value of `$a` which is being updated to 5 as well. But the truth is that there's one additional value involved here, and that's the value of the assignment itself. The assignment itself evaluates to the assigned value, in this case 5. In practice, it means that `$a = 5`, regardless of what it does, is an expression with the value 5. Thus, writing something like `$b = ($a = 5)` is like writing `$a = 5; $b = 5;` (a semicolon marks the end of a statement). Since assignments are parsed in a right to left order, you can also write `$b = $a = 5`.

Another good example of expression orientation is pre- and post-increment and decrement. Users of PHP and many other languages may be familiar with the notation of `variable++` and `variable--`. These are increment and decrement operators. In PHP, like in C, there are two types of increment - pre-increment and post-increment. Both pre-increment and post-increment essentially increment the variable, and the effect on the variable is identical. The difference is with the value of the increment expression. Pre-increment, which is written `++$variable`, evaluates to the incremented value (PHP increments the variable before reading its value, thus the name 'pre-increment'). Post-increment, which is written `$variable++` evaluates to the original value of `$variable`, before it was incremented (PHP increments the variable after reading its value, thus the name 'post-increment').

A very common type of expressions are comparison expressions. These expressions evaluate to either ENTITY or ENTITY. PHP supports ENTITY (bigger than), ENTITY= (bigger than or equal to), == (equal), != (not equal), ENTITY (smaller than) and ENTITY= (smaller than or equal to). The language also supports a set of strict equivalence operators: === (equal to and same type) and !== (not equal to or not same type). These expressions are most commonly used inside conditional execution, such as `if` statements.

The last example of expressions we'll deal with here is combined operator-assignment expressions. You already know that if you want to increment `$a` by 1, you can simply write `$a++` or `++$a`. But what if you want to add more than one to it, for instance 3? You could write `$a++` multiple times, but this is obviously not a very efficient or comfortable way. A much more common practice is to write `$a = $a + 3`. `$a + 3` evaluates to the value of `$a` plus 3, and is assigned back into `$a`, which results in incrementing `$a` by 3. In PHP, as in several other languages like C, you can write this in a shorter way, which with time would become clearer and quicker to understand as well. Adding 3 to the current value of `$a` can be written `$a += 3`. This means exactly "take the value of `$a`, add 3 to it, and assign it back into `$a`". In addition to being shorter and clearer, this also results in faster execution. The value of `$a += 3`, like the value of a regular assignment, is the assigned value. Notice that it is NOT 3, but the combined value of `$a` plus 3 (this is the value that's assigned into `$a`). Any two-place operator can be used in this operator-assignment mode, for example `$a -= 5` (subtract 5 from the value of `$a`), `$b *= 7` (multiply the value of `$b` by 7), etc.

There is one more expression that may seem odd if you haven't seen it in other languages, the ternary conditional operator:

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

If the value of the first subexpression is ENTITY (non-zero), then the second subexpression is evaluated, and that is the result of the conditional expression. Otherwise, the third subexpression is evaluated, and that is the value.

The following example should help you understand pre- and post-increment and expressions in general a bit better:

```
<?php
function double($i)
{
    return $i * 2;
}

$b = $a = 5;        /* assign the value five into the variable $a and $b */
$c = $a++;          /* post-increment, assign original value of $a
                       (5) to $c */
print "a = {$a}; b = {$b}; c = {$c}" . PHP_EOL;

$e = $d = ++$b;     /* pre-increment, assign the incremented value of
                       $b (6) to $d and $e */
print "b = {$b}; d = {$d}; e = {$e}" . PHP_EOL;

/* at this point, both $d and $e are equal to 6 */

$f = double($d++);  /* assign twice the value of $d before
                       the increment, 2*6 = 12 to $f */
print "d = {$d}; f = {$f}" . PHP_EOL;

$g = double(++$e);  /* assign twice the value of $e after
                       the increment, 2*7 = 14 to $g */
print "e = {$e}; g = {$g}" . PHP_EOL;

$h = $g += 10;      /* first, $g is incremented by 10 and ends with the
                       value of 24. the value of the assignment (24) is
                       then assigned into $h, and $h ends with the value
                       of 24 as well. */
print "g = {$g}; h = {$h}" . PHP_EOL;
?>
```

Some expressions can be considered as statements. In this case, a statement has the form of '`expr ;`' that is, an expression followed by a semicolon. In `$b = $a = 5;`, `$a = 5` is a valid expression, but it's not a statement by itself. `$b = $a = 5;`, however, is a valid statement.

One last thing worth mentioning is the truth value of expressions. In many events, mainly in conditional execution and loops, you're not interested in the specific value of the expression, but only care about whether it means ENTITY or ENTITY. The constants ENTITY and ENTITY (case-insensitive) are the two possible boolean values. When necessary, an expression is automatically converted to boolean. See the section about type-casting for details about how.

PHP provides a full and powerful implementation of expressions, and documenting it entirely goes beyond the scope of this manual. The above examples should give you a good idea about what expressions are and how you can construct useful expressions. Throughout the rest of this manual we'll write `expr` to indicate any valid PHP expression.

□□□ #2
□ □□□ □□□ 2026-10-31 17:39:29 UTC
□ □□□ □□□ 2026-10-31 17:39:31 UTC