Integers

- Syntax
- Integer overflow
- Integer division
- Converting to integer
  - From booleans
  - From floating point numbers
  - From strings
  - From NULL
  - From other types

An `int` is a number of the set ℤ = {..., -2, -1, 0, 1, 2, ...}.

ENTITY

Floating-point numbers, Arbitrary precision / BCMath, Arbitrary length integer / GMP

Syntax

`Int`s can be specified in decimal (base 10), hexadecimal (base 16), octal (base 8) or binary (base 2) notation. The negation operator can be used to denote a negative `int`.

To use octal notation, precede the number with a `0` (zero). As of PHP 8.1.0, octal notation can also be preceded with `0o` or `0O`. To use hexadecimal notation, precede the number with `0x`. To use binary notation, precede the number with `0b`.

As of PHP 7.4.0, integer literals may contain underscores (`_`) between digits, for better readability of literals. These underscores are removed by PHP's scanner.

Integer literals

```
<?php
$a = 1234; // decimal number
$a = 0123; // octal number (equivalent to 83 decimal)
$a = 0o123; // octal number (as of PHP 8.1.0)
$a = 0x1A; // hexadecimal number (equivalent to 26 decimal)
$a = 0b11111111; // binary number (equivalent to 255 decimal)
$a = 1_234_567; // decimal number (as of PHP 7.4.0)
?>
```

Formally, the structure for `int` literals is as of PHP 8.1.0 (previously, the `0o` or `0O` octal prefixes were not allowed, and prior to PHP 7.4.0 the underscores were not allowed):

```
decimal     : [1-9][0-9]*(_[0-9]+)*
            | 0

hexadecimal : 0[xX][0-9a-fA-F]+(_[0-9a-fA-F]+)*

octal       : 0[oO]?[0-7]+(_[0-7]+)*

binary      : 0[bB][01]+(_[01]+)*

integer     : decimal
            | hexadecimal
            | octal
            | binary
```

The size of an `int` is platform-dependent, although a maximum value of about two billion is the usual value (that's 32 bits signed). 64-bit platforms usually have a maximum value of about 9E18. PHP does not support unsigned `int`s. `int` size can be determined using the constant `PHP_INT_SIZE`, maximum value using the constant `PHP_INT_MAX`, and minimum value using the constant `PHP_INT_MIN`.

Integer overflow

If PHP encounters a number beyond the bounds of the `int` type, it will be interpreted as a `float` instead. Also, an operation which results in a number beyond the bounds of the `int` type will return a `float` instead.

Integer overflow

```
<?php
$large_number = 50000000000000000000;
var_dump($large_number);         // float(5.0E+19)

var_dump(PHP_INT_MAX + 1);       // 32-bit system: float(2147483648)
                                 // 64-bit system: float(9.2233720368548E+18)
?>
```

Integer division

There is no `int` division operator in PHP. To achieve this, use the `intdiv` function. `1/2` yields the `float` `0.5`. The value can be cast to an `int` to round it towards zero, or the `round` function provides finer control over rounding.

Divisions

```
<?php
var_dump(25/7);         // float(3.5714285714286)
var_dump((int) (25/7)); // int(3)
var_dump(round(25/7));  // float(4)
?>
```

Converting to integer

To explicitly convert a value to `int`, use the `(int)` cast. However, in most cases the cast is not needed, since a value will be automatically converted if an operator, function or control structure requires an `int` argument. A value can also be converted to `int` with the `intval` function.

If a `resource` is converted to an `int`, then the result will be the unique resource number assigned to the `resource` by PHP at runtime.

See also Type Juggling.

From booleans

ENTITY will yield `0` (zero), and ENTITY will yield `1` (one).

From floating point numbers

When converting from `float` to `int`, the number will be rounded *towards zero*. As of PHP 8.1.0, a deprecation notice is emitted when implicitly converting a non-integral ENTITY to an ENTITY which loses precision.

Casting from Float

```
<?php

function foo($value): int {
  return $value;
}

var_dump(foo(8.1)); // "Deprecated: Implicit conversion from float 8.1 to int loses precision"
as of PHP 8.1.0
var_dump(foo(8.1)); // 8 prior to PHP 8.1.0
var_dump(foo(8.0)); // 8 in both cases

var_dump((int) 8.1); // 8 in both cases
var_dump(intval(8.1)); // 8 in both cases
?>
```

If the float is beyond the boundaries of `int` (usually `+/- 2.15e+9 = 2^31` on 32-bit platforms and `+/- 9.22e+18 = 2^63` on 64-bit platforms), the result is undefined, since the `float` doesn't have enough precision to give an exact `int` result. As of PHP 8.5.0, an `E_WARNING` is emitted when a `float` that is not representable as an `int` is cast to one. Prior to PHP 8.5.0, no diagnostic at all was issued.

> [!NOTE] `NaN`, `Inf` and `-Inf` will always be zero when cast to `int`. As of PHP 8.5.0, casting them to `int` emits an `E_WARNING`, as does casting `NAN` to any

> other type.

> [!WARNING] Never cast an unknown fraction to `int`, as this can sometimes lead to unexpected results.
>
> ```
> <?php
> echo (int) ( (0.1+0.7) * 10 ); // echoes 7!
> ?>
> ```
>
> See also the warning about float precision.

From strings

If the string is numeric or leading numeric then it will resolve to the corresponding integer value, otherwise it is converted to zero (`0`).

From `NULL`

ENTITY is always converted to zero (`0`).

From other types

> [!CAUTION] The behaviour of converting to `int` is undefined for other types. Do *not* rely on any observed behaviour, as it can change without notice.

□□□ #2
□ □□□ □□□ 2026-10-31 17:47:42 UTC
□ □□□ □□□ 2026-10-31 17:47:43 UTC