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


Числовые строки »
« Числа с плавающей точкой

- Руководство по PHP
- Справочник языка
- Типы

Change language: Russian

# Строки

Строка (string) — это набор символов, в котором символ — это то же, что и байт. То есть PHP поддерживает набор только из 256 символов и поэтому не предлагает встроенную поддержку кодировки Unicode. Подробнее об этом рассказано в разделе «Подробные сведения о строковом типе».

> **Замечание**: В 32-битных сборках размер строки (string) ограничен 2 ГБ (2 147 483 647 байтов максимум).

## Синтаксис

Строковый литерал определяют четырьмя способами:

- одинарными кавычками
- двойными кавычками
- heredoc-синтаксисом
- nowdoc-синтаксисом

### Одинарные кавычки

Простейший способ определить строку — это заключить её в одинарные кавычки (символ `'`).

Чтобы записать внутри строки буквальную одинарную кавычку, её экранируют обратным слешем (`\`). Чтобы записать сам обратный слеш, его дублируют (`\\`). В остальных случаях обратный слеш будет обработан как буквальный обратный слеш: то есть последовательности вроде `\r` или `\n` не будут рассматриваться как управляющие, а будут выведены как записаны.

> **Замечание**: Переменные и управляющие последовательности служебных символов, заключённые в одинарные кавычки, — *не* обрабатываются, в отличие от синтаксиса двойных кавычек и heredoc.

```
<?php

echo 'Это — простая строка';

echo 'В строки также разрешено вставлять
символ новой строки, способом, которым записан этот текст, —
так делать нормально';

// Выводит: Однажды Арнольд сказал: "I'll be back"
echo 'Однажды Арнольд сказал: "I\'ll be back"';

// Выводит: Вы удалили C:\*.*?
echo 'Вы удалили C:\\*.*?';

// Выводит: Вы удалили C:\*.*?
echo 'Вы удалили C:\*.*?';

// Выводит: Это не будет развёрнуто: \n в новую строку
echo 'Это не будет развёрнуто: \n в новую строку';

// Выводит: Переменные $expand и $either также не разворачиваются
echo 'Переменные $expand и $either также не разворачиваются';

?>
```

### Двойные кавычки

Если строка заключена в двойные кавычки ("), PHP распознает следующие управляющие последовательности служебных символов:

| Управляющие последовательности | |
|---|---|
| **Последовательность** | **Значение** |

| | |
|---|---|
| `\n` | новая строка (LF или 0x0A (10) в ASCII) |
| `\r` | возврат каретки (CR или 0x0D (13) в ASCII) |
| `\t` | горизонтальная табуляция (HT или 0x09 (9) в ASCII) |
| `\v` | вертикальная табуляция (VT или 0x0B (11) в ASCII) |
| `\e` | escape-знак (ESC или 0x1B (27) в ASCII) |
| `\f` | подача страницы (FF или 0x0C (12) в ASCII) |
| `\\` | обратная косая черта |
| `\$` | знак доллара |
| `\"` | двойная кавычка |
| `\[0-7]{1,3}` | Восьмеричная запись: символ, код которого записан в восьмеричной нотации (т. е. `"\101" === "A"`), т. е. в виде последовательности символов, соответствующей регулярному выражению `[0-7]{1,3}`. В ситуации целочисленного переполнения (если символ не поместится в один байт), старшие биты будут без предупреждения отброшены (т. е. `"\400" === "\000"`) |
| `\x[0-9A-Fa-f]{1,2}` | Шестнадцатеричная система счисления: символ, код которого записан в шестнадцатеричной нотации (т. е. `"\x41" === "A"`), т. е. в виде последовательности символов, соответствующей регулярному выражению `[0-9A-Fa-f]{1,2}` |
| `\u{[0-9A-Fa-f]+}` | Стандарт Unicode: символ, код которого записан в нотации кодовых точек Unicode, т. е. в виде последовательности символов, соответствующей регулярному выражению `[0-9A-Fa-f]+`, которые будут отображены как строка в кодировке UTF-8. Последовательность необходимо заключать в фигурные скобки. Например: `"\u{41}" === "A"` |

Как и в строках в одинарных кавычках, экранирование другого символа выведет также и символ обратного слеша.

Наиболее важное свойство строк в двойных кавычках состоит в том, что имена переменных в них будут развёрнуты и обработаны. Подробнее об этом рассказано в разделе «Синтаксический анализ переменных».

**Heredoc**

Третий способ определения строк — это heredoc-синтаксис: `<<<`. Следом за этим оператором указывают идентификатор, а затем перевод строки. Затем идёт сама строка, за которой снова идёт тот же идентификатор, чтобы закрыть вставку.

Закрывающий идентификатор разрешено отбивать пробелами или символами табуляции, и тогда отступ будет удалён из каждой строки в блоке документа. До PHP 7.3.0 закрывающий идентификатор указывали *в самом начале* новой строки.

Кроме того, закрывающий идентификатор подчиняется тем же правилам именования, что и другие метки в PHP: содержит только буквенно-цифровые символы и подчёркивания, и не начинается с цифрового символа или символа подчёркивания.

**Пример #1 Базовый пример использования Heredoc-синтаксиса в PHP 7.3.0**

```
<?php

// без отступов
echo <<<END
a
b
c
\n
END;

// 4 отступа
echo <<<END
a
b
c
END;
```

Результат выполнения приведённого примера в PHP 7.3:

```
    a
```

```
    b
   c

  a
 b
c
```

Если закрывающий идентификатор смещён дальше хотя бы одной строки тела, будет выброшено исключение ParseError:

**Пример #2 Отступу закрывающего идентификатора нельзя отступать больше, чем другим строкам тела**

```
<?php

echo <<<END
a
b
c
END;
```

Результат выполнения приведённого примера в PHP 7.3:

```
PHP Parse error:  Invalid body indentation level (expecting an indentation level of at least 3) in example.php on line 4
```

Если закрывающий идентификатор отбит отступом, то в теле тоже разрешено указывать табуляции. Однако табуляциям и пробелам *не разрешено* смешиваться относительно отступа закрывающего идентификатора и тела (вплоть до закрывающего идентификатора). В каждом из этих случаев будет выброшено исключение ParseError. Эти ограничения на пробельные отступы добавили, потому что смешивание табуляций и пробелов для отступов вредно для разбора.

**Пример #3 Другой отступ для закрывающего идентификатора тела (пробелов)**

```
<?php

// Весь следующий код не работает.

// Другой отступ для закрывающего идентификатора (табуляций) тела (пробелов)
{
echo <<<END
a
END;
}

// Смешивание пробелов и табуляции в теле
{
echo <<<END
a
END;
}

// Смешивание пробелов и табуляции в закрывающем идентификаторе
{
echo <<<END
a
END;
}
```

Результат выполнения приведённого примера в PHP 7.3:

```
PHP Parse error:  Invalid indentation - tabs and spaces cannot be mixed in example.php line 8
```

За закрывающим идентификатором основной строки не обязательно ставить точку с запятой или новую строку. Например, начиная с PHP 7.3.0 разрешён следующий код:

**Пример #4 Продолжение выражения после закрывающего идентификатора**

```
<?php
```

```php
$values = [<<<END
a
b
c
END, 'd e f'];
var_dump($values);
```

Результат выполнения приведённого примера в PHP 7.3:

```
array(2) {
  [0] =>
  string(11) "a
  b
    c"
  [1] =>
  string(5) "d e f"
}
```

**Внимание**

Если закрывающий идентификатор найден в начале строки, даже если это часть слова, парсер примет его за закрывающий идентификатор и выбросит исключение ParseError.

**Пример #5 Закрывающий идентификатор в теле текста провоцирует исключение ParseError**

```php
<?php

$values = [<<<END
a
b
END ING
END, 'd e f'];
```

Результат выполнения приведённого примера в PHP 7.3:

```
PHP Parse error:  syntax error, unexpected identifier "ING", expecting "]" in example.php on line 6
```

Чтобы не возникало таких проблем, следуют несложному, но надёжному правилу: *не выбирать закрывающий идентификатор, который встречается в теле текста.*

**Внимание**

До PHP 7.3.0 строке с закрывающим идентификатором нельзя было содержать символов, кроме точки с запятой (`;`). То есть идентификатор *не разрешено вводить с отступом*, а пробелы или знаки табуляции нельзя вводить до или после точки с запятой. Учитывают также, что первым символом перед закрывающим идентификатором идёт символ новой строки, который определён в операционной системе. Например, в Unix-системах, включая macOS, это символ `\n`. После закрывающего идентификатора должна сразу начинаться новая строка.

Если это правило нарушено и закрывающий идентификатор не «чистый», он не будет считаться закрывающим, и PHP продолжит его поиск дальше. Если правильный закрывающий идентификатор так и не будет найден до конца текущего файла, то на последней строке возникнет ошибка синтаксического анализа.

**Пример #6 Пример неправильного синтаксиса, до PHP 7.3.0**

```php
<?php

class foo {
public $bar = <<<EOT
bar
EOT;
// Отступ перед закрывающим идентификатором недопустим
}

?>
```

**Пример #7 Пример правильного синтаксиса, даже до PHP 7.3.0**

```php
<?php

class foo {
public $bar = <<<EOT
bar
EOT;
}

?>
```

Heredoc с переменными нельзя использовать для инициализации свойств класса.

Heredoc-текст хотя и не заключён в двойные кавычки, ведёт себя как строка в двойных кавычках. То есть в heredoc кавычки не экранируют, но перечисленные управляющие коды по-прежнему разрешено указывать. Переменные разворачиваются, но в выражениях со сложными переменными внутри heredoc работают так же внимательно, как и при работе со строками.

**Пример #8 Пример определения heredoc-строки**

```php
<?php

$str = <<<EOD
Пример строки,
охватывающей несколько строк,
с использованием heredoc-синтаксиса.
EOD;

/* Более сложный пример с переменными. */
class foo
{
var $foo;
var $bar;

function __construct()
{
$this->foo = 'Foo';
$this->bar = array('Bar1', 'Bar2', 'Bar3');
}
}

$foo = new foo();
$name = 'Имярек';

echo <<<EOT
Меня зовут "$name". Я печатаю $foo->foo.
Теперь я вывожу {$foo->bar[1]}.
Это должно вывести заглавную букву 'A': \x41
EOT;

?>
```

Результат выполнения приведённого примера:

```
Меня зовут "Имярек". Я печатаю Foo.
Теперь, я вывожу Bar2.
Это должно вывести заглавную букву 'A': A
```

Heredoc-синтаксис разрешён также для передачи данных через аргументы функции:

**Пример #9 Пример heredoc-синтаксиса с аргументами**

```php
<?php

var_dump(array(<<<EOD
foobar!
```

```
EOD
));

?>
```

Разрешено инициализировать статические переменные и свойства или константы класса в heredoc-синтаксисе:

**Пример #10 Инициализация статических переменных heredoc-синтаксисом**

```
<?php

// Статические переменные
function foo()
{
static $bar = <<<LABEL
Здесь ничего нет...
LABEL;
}

// Константы/свойства класса
class foo
{
const BAR = <<<FOOBAR
Пример использования константы
FOOBAR;

public $baz = <<<FOOBAR
Пример использования поля
FOOBAR;
}

?>
```

Разрешено также окружать heredoc-идентификатор двойными кавычками:

**Пример #11 Двойные кавычки в heredoc**

```
<?php

echo <<<"FOOBAR"
Привет, мир!
FOOBAR;

?>
```

### Nowdoc

Nowdoc — это то же для строк в одинарных кавычках, что и heredoc для строк в двойных кавычках. Nowdoc похож на heredoc, но внутри него *не выполняются подстановки*. Конструкция легко встраивает PHP-код или другие большие блоки текста без предварительного экранирования. В этом он отчасти похож на SGML-конструкцию `<![CDATA[ ]]>`, в том, что он объявляет блок текста, который не требует обработки.

Nowdoc задают той же последовательностью символов `<<<`, что и в heredoc, но следующий за ней идентификатор берут в одинарные кавычки, например, `<<<'EOT'`. Условия, которые распространяются на идентификаторы heredoc-синтаксиса, действительны также и для синтаксиса nowdoc, а больше остальных те, что относятся к закрывающему идентификатору.

**Пример #12 Пример nowdoc-синтаксиса**

```
<?php

echo <<<'EOD'
Пример текста,
занимающего несколько строк,
```

```
написанного синтаксисом nowdoc. Обратные слеши выводятся без обработки,
например, \\ и \'.
EOD;
```

Результат выполнения приведённого примера:

```
Пример текста,
занимающего несколько строк,
написанного синтаксисом nowdoc. Обратные слеши выводятся без обработки,
например, \\ и \'.
```

**Пример #13 Nowdoc с переменными в строках с двойными кавычками**

```
<?php

/* Усложнённый пример с переменными. */
class foo
{
public $foo;
public $bar;

function __construct()
{
$this->foo = 'Foo';
$this->bar = array('Bar1', 'Bar2', 'Bar3');
}
}

$foo = new foo();
$name = 'Имярек';

echo <<<'EOT'
Меня зовут "$name". Я печатаю $foo->foo.
Теперь я печатаю {$foo->bar[1]}.
Это не должно вывести заглавную 'A': \x41
EOT;

?>
```

Результат выполнения приведённого примера:

```
Меня зовут "$name". Я печатаю $foo->foo.
Теперь я печатаю {$foo->bar[1]}.
Это не должно вывести заглавную 'A': \x41
```

**Пример #14 Пример со статичными данными**

```
<?php

class foo {
public $bar = <<<'EOT'
bar
EOT;
}

?>
```

**Синтаксический анализ переменных**

Если строка указывается в двойных кавычках, либо синтаксисом heredoc, переменные внутри неё обрабатываются.

В PHP предусмотрели два вида синтаксиса для указания переменных в строках: простой и сложный. Простым синтаксисом пользуются чаще, с ним легко встраивать переменную, значение массива (array) или свойство объекта (object) с минимумом усилий.

Сложный синтаксис легко определить по фигурным скобкам, которые окружают выражение.

**Простой синтаксис**

Если интерпретатор встречает знак доллара ($), он захватывает как можно больше символов, чтобы сформировать правильное имя переменной. Если нужно точно определить конец имени, имя переменной берут в фигурные скобки.

```
<?php

$juice = "apple";

echo "He drank some $juice juice." . PHP_EOL;

// Непредусмотрительно. Символ «s» — корректный символ для имени переменной, поэтому в этом примере он относится к
переменной $juces, но не $juice.
echo "He drank some juice made of $juices." . PHP_EOL;

// Укажем границы переменной, взяв её в фигурные скобки.
echo "He drank some juice made of {$juice}s.";

?>
```

Результат выполнения приведённого примера:

```
He drank some apple juice.
He drank some juice made of .
He drank some juice made of apples.
```

Аналогично будет проанализирован индекс массива (array) или свойство объекта (object). В индексах массива закрывающая квадратная скобка (]) означает конец определения индекса. На свойства объекта распространяются те же правила, что и на простые переменные.

**Пример #15 Пример простого синтаксиса**

```
<?php

$juices = array("apple", "orange", "koolaid1" => "purple");

echo "He drank some $juices[0] juice.".PHP_EOL;
echo "He drank some $juices[1] juice.".PHP_EOL;
echo "He drank some $juices[koolaid1] juice.".PHP_EOL;

class people {
public $john = "John Smith";
public $jane = "Jane Smith";
public $robert = "Robert Paulsen";

public $smith = "Smith";
}

$people = new people();

echo "$people->john drank some $juices[0] juice.".PHP_EOL;
echo "$people->john then said hello to $people->jane.".PHP_EOL;
echo "$people->john's wife greeted $people->robert.".PHP_EOL;
echo "$people->robert greeted the two $people->smiths."; // Не сработает

?>
```

Результат выполнения приведённого примера:

```
He drank some apple juice.
He drank some orange juice.
He drank some purple juice.
John Smith drank some apple juice.
John Smith then said hello to Jane Smith.
John Smith's wife greeted Robert Paulsen.
Robert Paulsen greeted the two .
```

В PHP 7.1.0 добавлена поддержка *отрицательных* числовых индексов.

**Пример #16 Отрицательные числовые индексы**

```php
<?php

$string = 'string';
echo "Символ с индексом -2 равен $string[-2].", PHP_EOL;
$string[-3] = 'o';
echo "Изменение символа на позиции -3 на «o» даёт следующую строку: $string.", PHP_EOL;

?>
```

Результат выполнения приведённого примера:

```
Символ с индексом -2 равен n.
Изменение символа на позиции -3 на «o» даёт следующую строку: strong
```

Для выражений, которые сложнее этих, лучше пользоваться сложным синтаксисом.

#### Сложный (фигурный) синтаксис

Он называется сложным не из-за сложности синтаксиса, а только потому, что разрешает писать сложные выражения.

Скалярная переменная, элемент массива или отображаемое в строку свойство объекта разрешено указывать в строке этим синтаксисом. Выражение записывается как и вне строки, а затем берётся в фигурные скобки: `{` и `}`. Поскольку знак `{` невозможно экранировать, этот синтаксис будет распознаваться только тогда, когда знак `$` идёт непосредственно за знаком `{`. Чтобы получить литерал `{$`, знак доллара экранируют `{\$`. Поясняющие примеры:

```php
<?php

// Показываем все ошибки
error_reporting(E_ALL);

$great = 'здорово';

// Не работает, выводит: Это { здорово}
echo "Это { $great}";

// Работает, выводит: Это здорово
echo "Это {$great}";

// Работает
echo "Этот квадрат шириной {$square->width}00 сантиметров.";

// Работает, ключи, взятые в кавычки, работают только с синтаксисом фигурных скобок
echo "Это работает: {$arr['key']}";

// Работает
echo "Это работает: {$arr[4][3]}";

// Это неверно по той же причине, что и $foo[bar] вне
// строки. Говоря по-другому, это по-прежнему работает,
// но поскольку PHP сначала ищет константу foo, это вызовет
// ошибку уровня E_NOTICE (неопределённая константа).
echo "Это неправильно: {$arr[foo][3]}";

// Работает. При обращении к многомерным массивам внутри
// строк указывают фигурные скобки
echo "Это работает: {$arr['foo'][3]}";

// Работает.
echo "Это работает: " . $arr['foo'][3];
```

```php
echo "Это тоже работает: {$obj->values[3]->name}";

echo "Это значение переменной с именем $name: {${$name}}";

echo "Это значение переменной с именем, которое возвращает функция getName(): {${getName()}}";

echo "Это значение переменной с именем, которое возвращает \$object->getName(): {${$object->getName()}}";

// Не работает, выводит: Это то, что возвращает функция getName(): {getName()}
echo "Это то, что возвращает функция getName(): {getName()}";

// Не работает, выводит: C:\folder\{fantastic}.txt
echo "C:\folder\{$great}.txt"

// Работает, выводит: C:\folder\fantastic.txt
echo "C:\\folder\\{$great}.txt"

?>
```

Разрешено получать доступ к свойствам класса, указывая переменные внутри строк, записанных этим синтаксисом.

```php
<?php

class foo {
var $bar = 'I am bar.';
}

$foo = new foo();
$bar = 'bar';
$baz = array('foo', 'bar', 'baz', 'quux');
echo "{$foo->$bar}\n";
echo "{$foo->{$baz[1]}}\n";

?>
```

Результат выполнения приведённого примера:

```
I am bar.
I am bar.
```

> **Замечание**:
>
> Значение внутри литерала `{$}`, к которому получают доступ из функций, вызовов методов, статических переменных класса и констант класса, интерпретируется как имя переменной в области, в которой определена строка. Синтаксис одинарных фигурных скобок (`{}`) не будет работать для доступа к значениям функций, методов, констант классов или статических переменных класса.

```php
<?php

// Показываем все ошибки
error_reporting(E_ALL);

class beers {
const softdrink = 'rootbeer';
public static $ale = 'ipa';
}

$rootbeer = 'A & W';
$ipa = 'Alexander Keith\'s';

// Это работает, выводит: Я бы хотел A & W
echo "Я бы хотел {${beers::softdrink}}\n";

// Это тоже работает, выводит: Я бы хотел Alexander Keith's
```

```php
echo "Я бы хотел {${beers::$ale}}\n";

?>
```

**Доступ и изменение символа в строке**

Чтобы получить доступ и изменить символ в строке, нужно в квадратных скобках после переменной определить смещение искомого символа относительно начала строки начиная с нуля, например, *$str[42]*. Для этого о строке думают как о массиве символов. Чтобы получить или заменить больше одного символа, вызывают функции substr() и substr_replace().

> **Замечание**: Начиная с PHP 7.1.0 поддерживаются отрицательные значения смещения. Они задают смещение с конца строки. Раньше отрицательные смещение вызывали ошибку уровня **E_NOTICE** при чтении (возвращая пустую строку) или **E_WARNING** при записи (оставляя строку без изменений).

> **Замечание**: До PHP 8.0.0 доступ к символам в строках (string) получали, указывая фигурные скобки, например *$str{42}*. Синтаксис фигурных скобок устарел с PHP 7.4.0 и не поддерживается с PHP 8.0.0.

**Внимание**

Попытка записи в смещение за границами строки дополнит строку пробелами до этого смещения. Нецелочисленные типы преобразуются в целочисленные. Неверный тип смещения выдаст ошибку уровня **E_WARNING**. При добавлении в смещение строки новых символов присвоится только первый символ (байт). Начиная с PHP 7.1.0 присваивание пустой строки вызовет фатальную ошибку. Раньше присваивался нулевой байт (NULL).

**Внимание**

Внутренне строки PHP представлены массивами байтов. Поэтому доступ или изменение строки по смещению небезопасны для многобайтовых данных и выполняются только со строками в однобайтных кодировках, например ISO-8859-1.

> **Замечание**: Начиная с PHP 7.1.0 попытка указать оператор пустого индекса на пустой строке выдаст фатальную ошибку. Раньше пустая строка преобразовывалась в массив без предупреждения.

**Пример #17 Примеры строк**

```php
<?php

// Получим первый символ строки
$str = 'This is a test.';
$first = $str[0];

// Получим третий символ строки
$third = $str[2];

// Получим последний символ строки
$str = 'This is still a test.';
$last = $str[strlen($str)-1];

// Изменим последний символ строки
$str = 'Look at the sea';
$str[strlen($str)-1] = 'e';

?>
```

Смещение в строке задают либо целым числом, либо целочисленной строкой, иначе будет выдано предупреждение.

**Пример #18 Пример недопустимого смещения строки**

```php
<?php

$str = 'abc';

var_dump($str['1']);
var_dump(isset($str['1']));
```

```php
var_dump($str['1.0']);
var_dump(isset($str['1.0']));

var_dump($str['x']);
var_dump(isset($str['x']));

var_dump($str['1x']);
var_dump(isset($str['1x']));

?>
```

Результат выполнения приведённого примера:

```
string(1) "b"
bool(true)

Warning: Illegal string offset '1.0' in /tmp/t.php on line 7
string(1) "b"
bool(false)

Warning: Illegal string offset 'x' in /tmp/t.php on line 9
string(1) "a"
bool(false)
string(1) "b"
bool(false)
```

> **Замечание**:
>
> Доступ к переменным других типов (не включая массивы, а также объекты, реализующие соответствующие интерфейсы) через операторы `[]` или `{}` без предупреждения возвращает **`null`**.

> **Замечание**:
>
> Доступ к символам в строковых литералах получают через операторы `[]` или `{}`.

> **Замечание**:
>
> Доступ к символам в строковых литералах через оператор `{}` объявлен устаревшим в PHP 7.4 и удалён в PHP 8.0.

## Полезные функции и операторы

Строки разрешено объединять оператором «.» (точка). Обратите внимание, оператор сложения «+» здесь *не работает*. Подробнее об этом рассказано в разделе «Строковые операторы».

В языке предусмотрен ряд полезных функций для манипулирования строками.

Общие функции описаны в разделе «Функции для работы со строками», а для расширенного поиска и замены — «Функции Perl-совместимых регулярных выражений».

Предусмотрены также функции для работы с URL и функции шифрования или дешифрования строк (Sodium и Hash).

Наконец, смотрите также функции символьных типов.

## Преобразование в строку

Значение преобразовывают в строку приведением через оператор `(string)` или функцией strval(). В выражениях, в которых требуется строка, преобразование выполняется автоматически. Это выполняется во время вывода через языковые конструкции echo или print, либо когда значение переменной сравнивается со строкой. Разделы руководства «Типы» и «Манипуляции с типами», прояснят сказанное ниже. Смотрите также описание функции settype().

Значение bool **`true`** преобразовывается в строку «`1`», а логическое значение **`false`** преобразовывается в «» (пустую строку). Такое поведение допускает преобразование значения в обе стороны — из логического типа в строковый и наоборот.

Целое число (int) или число с плавающей точкой (float) преобразовывается в строку, которая будет представлять число в текстовом виде (включая экспоненциальную часть для чисел с плавающей точкой). Большие числа с

плавающей точкой преобразовываются в экспоненциальную запись (`4.1E+6`).

> **Замечание**:
>
> Начиная с PHP 8.0.0 в качестве разделителя дробной части в числах с плавающей точкой разрешено использовать только точку («.»). До PHP 8.0.0 символ десятичной точки определялся в настройках языкового стандарта скрипта (категория LC_NUMERIC). Смотрите функцию setlocale().

Массивы преобразовываются в строку «`Array`». Поэтому конструкции echo или print не умеют без помощи функций отображать содержимое массива (array). Чтобы просмотреть отдельный элемент, пользуются синтаксисом `echo $arr['foo']`. Ниже будет рассказано о том, как отобразить или просмотреть всё содержимое.

Для преобразования объекта (`object`) в строку (string) определяют магический метод __toString.

Ресурс (resource) преобразовывается в строку (string) вида «`Resource id #1`», где `1` — это номер ресурса, который PHP назначает ресурсу (resource) во время исполнения кода. И хотя она уникальна для текущего запуска скрипта (т. е. веб-запроса или CLI-процесса) и не будет использована повторно для этого ресурса, не стоит полагаться на эту строку, потому что её могут изменить в будущем. Тип ресурса можно получить вызовом функции get_resource_type().

Значение **`null`** всегда преобразовывается в пустую строку.

Как указано выше, прямое преобразование в строку массивов, объектов или ресурсов не даёт полезной информации о значении, кроме типа. Более эффективные инструменты вывода значений для отладки этих типов — это функции print_r() и var_dump().

Бóльшая часть значений в PHP преобразуема в строку для постоянного хранения. Этот метод преобразования называется сериализацией. Сериализуют значения функцией serialize().

## Подробные сведения о строковом типе

Строковый тип (string) в PHP реализован в виде массива байтов и целочисленного значения, содержащего длину буфера. В этой структуре нет информации о том, как преобразовывать байты в символы, эту задачу решает программист. Нет ограничений на значения, из которых состоит строка, например, байт со значением `0` (NUL-байт) разрешён где угодно в строке (однако рекомендовано учитывать, что ряд функций, которые в этом руководстве названы «бинарно-небезопасными», передают строки библиотекам, которые игнорируют данные после NUL-байта).

Такая природа строкового типа объясняет, почему в PHP нет отдельного типа «byte» — строки выполняют эту роль. Функции, которые не возвращают текстовых данных, — например, произвольный поток данных, считываемый из сетевого сокета, — по-прежнему возвращают строки.

С учётом того, что PHP не диктует конкретную кодировку для строк, может возникнуть вопрос: Как тогда кодируются строковые литералы? Например, строка «á» эквивалентна «`\xE1`» (ISO-8859-1), «`\xC3\xA1`» (UTF-8, форма нормализации C), «`\x61\xCC\x81`» (UTF-8, форма нормализации D) или другому возможному представлению? Ответ такой: строка будет закодирована способом, которым она закодирована в файле скрипта. Поэтому, если скрипт записан в кодировке ISO-8859-1, то и строка будет закодирована в ISO-8859-1 и т. д. Однако это правило не выполняется при включённом режиме Zend Multibyte: скрипт записывают в произвольной кодировке, объявляя её или полагаясь на автоопределение, а затем конвертируют в конкретную внутреннюю кодировку, которая и будет использована для строковых литералов. Учтите, что на кодировку скрипта (или на внутреннюю кодировку, если включён режим Zend Multibyte) накладывается ряд ограничений: почти в каждом случае эта кодировка должна быть надмножеством кодировки ASCII, например, UTF-8 или ISO-8859-1. Учтите также, что кодировки, зависящие от состояния, где одни и те же значения байтов допустимы в начальном и не начальном состоянии сдвига, создают риск проблем.

Строковые функции, чтобы быть полезными, пробуют предположить кодировку строки. Единство в этом вопросе не помешало бы, но PHP-функции работают с текстом по-разному:

- Одни — предполагают, что строка закодирована в какой-то однобайтовой кодировке, но для корректной работы им не нужно интерпретировать байты как конкретные символы. Сюда попадают функции вроде substr(), strpos(), strlen() и strcmp(). Другой способ мышления об этих функциях — представлять, что они оперируют буферами памяти, т. е. работают непосредственно с байтами и их смещениями.
- Другим — передаётся кодировка строки или они принимают значение по умолчанию, если кодировку не передали. Это относится к функции htmlentities() и большей части функций модуля mbstring.
- Третьи — работают с текущими настройками локали (смотрите setlocale()), но оперируют побайтово.
- Наконец четвёртые — предполагают, что строка использует конкретную кодировку, обычно UTF-8. Сюда попадает бóльшая часть функций из модулей intl и PCRE (для последнего — только при указании модификатора `u`).

В конечном счёте, программа будет работать с кодировкой Unicode правильно, если старательно избегать функций, которые не будут работать с Unicode-строками или повредят данные, и вызывать вместо них те, которые ведут себя корректно, обычно это функции из модулей intl и mbstring. Однако работа с функциями, которые умеют обрабатывать Unicode, — это только начало. Независимо от того, какие функции предлагает язык, рекомендовано знать спецификацию Unicode. Например, программа, которая предполагает существование только прописных и строчных букв, делает неверное предположение.

＋add a note

## User Contributed Notes 12 notes

up
down
109
***gtisza at gmail dot com*** ¶
**12 years ago**

```
The documentation does not mention, but a closing semicolon at the end of the heredoc is actually interpreted as a real semicolon, and as such, sometimes leads to syntax errors.

This works:

<?php
$foo = <<<END
abcd
END;
?>

This does not:

<?php
foo(<<<END
abcd
END;
);
// syntax error, unexpected ';'
?>

Without semicolon, it works fine:

<?php
foo(<<<END
abcd
END
);
?>
```

up
down
22
***BahmanMD*** ¶
**1 year ago**

```
In PHP 8.2 using ${var} in strings is deprecated, use {$var} instead:

<?php
$juice = "apple";

// Valid. Explicitly specify the end of the variable name by enclosing it in braces:
echo "He drank some juice made of {$juice}s.";
?>
```

up
down
24
***lelon at lelon dot net*** ¶

**19 years ago**

You can use the complex syntax to put the value of both object properties AND object methods inside a string. For example...

```
<?php
class Test {
public $one = 1;
public function two() {
return 2;
}
}
$test = new Test();
echo "foo {$test->one} bar {$test->two()}";
?>
```

Will output "foo 1 bar 2".

However, you cannot do this for all values in your namespace. Class constants and static properties/methods will not work because the complex syntax looks for the '$'.

```
<?php
class Test {
const ONE = 1;
}
echo "foo {Test::ONE} bar";
?>
```

This will output "foo {Test::one} bar". Constants and static properties require you to break up the string.

up
down
20

***og at gams dot at ¶***

**16 years ago**

easy transparent solution for using constants in the heredoc format:

```
DEFINE('TEST','TEST STRING');

$const = get_defined_constants();

echo <<<END
{$const['TEST']}
END;

Result:
TEST STRING
```

up
down
9

***Ray.Paseur sometimes uses Gmail ¶***

**5 years ago**

```
md5('240610708') == md5('QNKCDZO')
```

This comparison is true because both md5() hashes start '0e' so PHP type juggling understands these strings to be scientific notation. By definition, zero raised to any power is zero.

up
down
13

***steve at mrclay dot org ¶***

**15 years ago**

Simple function to create human-readably escaped double-quoted strings for use in source code or when debugging strings with newlines/tabs/etc.

```
<?php
function doubleQuote($str) {
$ret = '"';
for ($i = 0, $l = strlen($str); $i < $l; ++$i) {
$o = ord($str[$i]);
```

```
if ($o < 31 || $o > 126) {
switch ($o) {
case 9: $ret .= '\t'; break;
case 10: $ret .= '\n'; break;
case 11: $ret .= '\v'; break;
case 12: $ret .= '\f'; break;
case 13: $ret .= '\r'; break;
default: $ret .= '\x' . str_pad(dechex($o), 2, '0', STR_PAD_LEFT);
}
} else {
switch ($o) {
case 36: $ret .= '\$'; break;
case 34: $ret .= '\"'; break;
case 92: $ret .= '\\\\'; break;
default: $ret .= $str[$i];
}
}
}
return $ret . '"';
}
?>
```

up
down
7

***php at richardneill dot org ¶***

**10 years ago**

Leading zeroes in strings are (least-surprise) not treated as octal.
Consider:

```
$x = "0123" + 0;
$y = 0123 + 0;
echo "x is $x, y is $y"; //prints "x is 123, y is 83"
```

in other words:
* leading zeros in numeric literals in the source-code are interpreted as "octal", c.f. strtol().
* leading zeros in strings (eg user-submitted data), when cast (implicitly or explicitly) to integer are ignored, and considered as decimal, c.f. strtod().

up
down
8

***atnak at chejz dot com ¶***

**19 years ago**

Here is a possible gotcha related to oddness involved with accessing strings by character past the end of the string:

```
$string = 'a';

var_dump($string[2]); // string(0) ""
var_dump($string[7]); // string(0) ""
$string[7] === ''; // TRUE
```

It appears that anything past the end of the string gives an empty string.. However, when E_NOTICE is on, the above examples will throw the message:

```
Notice: Uninitialized string offset: N in FILE on line LINE
```

This message cannot be specifically masked with @$string[7], as is possible when $string itself is unset.

```
isset($string[7]); // FALSE
$string[7] === NULL; // FALSE
```

Even though it seems like a not-NULL value of type string, it is still considered unset.

up
down
3

***necrodust44 at gmail dot com ¶***

**9 years ago**

String conversion to numbers.

Unfortunately, the documentation is not correct.

«The value is given by the initial portion of the string. If the string starts with valid numeric data, this will be the value used. Otherwise, the value will be 0 (zero).»

It is not said and is not shown in examples throughout the documentation that, while converting strings to numbers, leading space characters are ignored, like with the strtod function.

```
<?php
echo " \v\f \r 1234" + 1; // 1235
var_export ("\v\f \r 1234" == "1234"); // true
?>
```

However, PHP's behaviour differs even from the strtod's. The documentation says that if the string contains a "e" or "E" character, it will be parsed as a float, and suggests to see the manual for strtod for more information. The manual says

«A hexadecimal number consists of a "0x" or "0X" followed by a nonempty sequence of hexadecimal digits possibly containing a radix character, optionally followed by a binary exponent. A binary exponent consists of a 'P' or 'p', followed by an optional plus or minus sign, followed by a nonempty sequence of decimal digits, and indicates multiplication by a power of 2.»

But it seems that PHP does not recognise the exponent or the radix character.

```
<?php
echo "0xEp4" + 1; // 15
?>
```

strtod also uses the current locale to choose the radix character, but PHP ignores the locale, and the radix character is always 2E. However, PHP uses the locale while converting numbers to strings.

With strtod, the current locale is also used to choose the space characters, I don't know about PHP.

up

down

5

***chAlx at findme dot if dot u dot need ¶***

**15 years ago**

To save Your mind don't read previous comments about dates ;)

When both strings can be converted to the numerics (in ("$a" > "$b") test) then resulted numerics are used, else FULL strings are compared char-by-char:

```
<?php
var_dump('1.22' > '01.23'); // bool(false)
var_dump('1.22.00' > '01.23.00'); // bool(true)
var_dump('1-22-00' > '01-23-00'); // bool(true)
var_dump((float)'1.22.00' > (float)'01.23.00'); // bool(false)
?>
```

up

down

0

***greenbluemoonlight at gmail dot com ¶***

**3 years ago**

```
<?php
\\Example # 10 Simple Syntax - Solution for the last "echo" line.

class people {
public $john = "John Smith";
public $jane = "Jane Smith";
```

```
public $robert = "Robert Paulsen";

public $smith = "Smith";
}

$people = new people();

echo "$people->john then said hello to $people->jane.".PHP_EOL;
echo "$people->john's wife greeted $people->robert.".PHP_EOL;
echo "$people->robert greeted the two $people->smiths";
\\Won't work
\\Outputs: Robert Paulsen greeted the two

/**Solution:**\

echo "$people->robert greeted the two $people->smith\x08s";

\\Will work
\\Outputs: Robert Paulsen greeted the two Smiths

?>
```

up
down
2

***headden at karelia dot ru ¶***

**14 years ago**

Here is an easy hack to allow double-quoted strings and heredocs to contain arbitrary expressions in curly braces syntax, including constants and other function calls:

```
<?php

// Hack declaration
function _expr($v) { return $v; }
$_expr = '_expr';

// Our playground
define('qwe', 'asd');
define('zxc', 5);

$a=3;
$b=4;

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

// Usage
echo "pre {$_expr(1+2)} post\n"; // outputs 'pre 3 post'
echo "pre {$_expr(qwe)} post\n"; // outputs 'pre asd post'
echo "pre {$_expr(c($a, $b)+zxc*2)} post\n"; // outputs 'pre 17 post'

// General syntax is {$_expr(...)}
?>
```

＋add a note