С.Н. Сорокин, Д.В. Федосов

РАЗРАБОТКА WEB-ПРИЛОЖЕНИЙ

ЧАСТЬ 1

Учебно-методическое пособие

Предисловие

При освоении дисциплины «Языки программирования» обучающимся необходимо изучить язык *PHP*, позволяющий быстро разрабатывать современные web-приложения различной степени сложности. В первой части пособия рассматриваются основы языка *PHP*, такие как: типовая компоновка программы, базовые синтаксические конструкции, использование функций в *PHP*, применение массивов и строк, математические функции, работа с файловой системой, использование регулярных выражений, обработка даты и времени, расширенные возможности языка и основы объектно-ориентированного программирования в *PHP*.

Для изучения пособия требуются базовые навыки по разработке *web*-интерфейсов, получаемые при изучении дисциплины «Разработка *web*-интерфейсов». Предполагается, что после освоения настоящего пособия слушатели смогут разрабатывать web-приложения, а при необходимости разбираться с исходным кодом и вносить изменения в приложения, написаные с использованием типовых систем управления контентом.

Лекционные занятия по дисциплине целесообразно проводить в аудитории, оборудованной мультимедиа оборудованием с демонстрацией использования различных инструментальных средств по разработке web-приложений. Для стимулирования обучающихся в регулярном освоении материала желательно регулярно проводить краткий устный опрос или тестирование по пройденным темам; а в начале каждого практического занятия – проводить контрольный письменный опрос, с последующей оценкой ответов и выставлением оценок в журнал учебной группы. Кроме того, в журнале выставляются оценки за все лабораторные работы и за контрольную работу. Совокупность всех оценок помогает в ходе проведения зачёта объективно оценить уровень усвоения учебного материала каждым обучающимся.

Настоящее учебно-методическое пособие состоит из десяти глав, соответствующих основным темам дисциплины.

В первой главе «Введение в *PHP*» необходимо получить общее представление о языке *PHP*. Рассматриваются: характеристика языка *PHP*, стандарты оформления кода *PSR*, компоновка типовой программы, базовые синтаксические конструкции и операторы изменения хода выполнения программы. Особое внимание следует уделить изучению типовой структуры программ на *PHP* и базовым синтаксическим конструкциям.

При изучении второй главы «Использование функций в *PHP*» слушатели должны в полной мере усвоить синтаксис описания функций, области видимости переменных, способы передачи параметров и основы рекурсивного вызова функций. Наиболее важно при проработке материала главы научиться правильно определять код, который необходимо вынести в функцию, понимать особенности работы с параметрами различных типов.

В третьей главе «Массивы» рассматриваются основные способы работы с массивами в *PHP*, такие как: отличия ассоциативного массива и списка, доступные операции над массивами, подходы к изменению порядков элементов, применение массивов как множеств. В этой теме обучающимся необходимо уяснить особенности правильного применения массивов различного вида в *PHP*, их преимущества и недостатки.

В четвертой главе «Строки» представлена необходимая информация об особенностях использования строкового типа в языке *PHP*. Рассмотрены вопросы работы с различными кодировками строк и стандартными операциями со строками.

В пятой главе «Математические функции» изложены способы корректного применения математических операций в языке *PHP*, такие как: округление чисел, генерация случайных чисел, преобразования между системами счисления, поиск минимума и максимума, различные типы функций.

В шестой главе «Работа с файлами и каталогами» представлены подходы к взаимодействию с файловой системой из языка *PHP*. Рассмотрены следующие вопросы: открытие и закрытие файлов, их чтение и запись, работа с путями, работа с файлами и каталогами. Также рассматриваются блокировки файлов, управление правами доступа и взаимодействие с внешними ресурсами.

В седьмой главе «Регулярные выражения в формате *PCRE*» рассматриваются вопросы работы с регулярными выражениями в языке *PHP*, а именно: синтаксис *PCRE*, функции для работы с регулярными выражениями, использование карманов и правильные приемы работы с регулярными выражениями.

В восьмой главе «Работа с датой и временем в *PHP*» представлена информации о формате времени в *PHP*, строковых представлениях дат, получении даты в виде массива, проверки дат, календарных классах в *PHP* и других аспектах работы с временем и датой.

В девятой главе «Расширенные возможности языка *PHP*» излагаются вопросы взаимодействия с внешними программами, управления интерпретатором *PHP* и генерации кода во время выполнения программы.

В десятой главе «Объектно-ориентированное программирование в *PHP*» рассматриваются основы использования классовых структур для упрощения разработки на языке *PHP*. Излагаются следующие вопросы: создание, инициализация и разрушение объектов, доступ к членам класса, клонирование и сериализация объектов, основы наследования (включение, расширение класса, запрет наследования и позднее статическое связывание), полиморфизм (абстрактные методы, интерфейсы, трейты) и применение пространства имен.

# Глава 1. Введение в PHP

## 1.1. Характеристика языка PHP

### 1.1.1. История появления и развития языка

В 1995 году датский программист Расмус Лердорф создал набор скриптов на *Perl/CGI* для вывода и учёта посетителей его онлайн-резюме, обрабатывающий шаблоны *HTML*-документов. Функциональности и быстроты *Perl* перестало хватать, поэтому Лердорф разработал с использованием языка *C* новый интерпретатор шаблонов *PHP/FI.* *PHP/FI* включал базовую функциональность современного *PHP*: оформление переменных в стиле *Perl*, автоматическую обработку форм и встраиваемость в *HTML*-текст и многое другое. Новорождённый язык отличался от своего прародителя более простым и ограниченным синтаксисом.

В 1997 году после длительного бета-тестирования вышла вторая версия обработчика, написанного на *C* – *PHP/FI* 2.0. Её использовали около 1% (приблизительно 50 тысяч) всех интернет-доменов мира.

Версия *PHP* 3.0, разработанная на *C* двумя израильскими программистами Энди Гутмансом и Зеевом Сураски, была построена на модульной основе, позволяющей сторонним командам создавать свои расширения, снабжающими язык новыми возможностями по взаимодействию с базами данных, обработке изображений, реализации алгоритмов шифрования. Эта концепция существует и по сей день.

В 1999 году была выпущена четвертая версия языка *PHP* 4.0 на движке, названном *Zend Engine*, обновленные версии которого используются до настоящего времени, определяя все достоинства и недостатки языка.

В 2005 году выходит следующая версия **-** *PHP* 5, построенная на концепции объектно-ориентированного программирования.

Успехи социальных сетей в области создания альтернативных реализаций *PHP* побудило сообщество к выпуску новой, более эффективной версии *PHP* 7 на новом движке *Zend Engine* 3, выпуск которой состоялся в декабре 2015 года. Команда разработчиков смогла ускорить движок *PHP* 7 в два раза, на фоне снижения потребления оперативной памяти. В версии *PHP* 7 значительно улучшена поддержка 64-битных операционных систем.

### 1.1.2. Краткая характеристика PHP

*PHP*, расшифровывающийся как "*PHP*: *Hypertext Preprocessor*", является распространенным интерпретируемым языком общего назначения с открытым исходным кодом. Язык обладает следующими характеристиками:

* мультипарадигменный (функциональный, процедурный, объектно-ориентированный);
* тип исполнения – интерпретатор компилирующего типа (*Zend Engine* 3), в отличие от компилируемых языков программирования, где весь текст программы, перед запуском, анализируется и транслируется в машинный или байт-код, без её выполнения, интерпретатор построчно анализирует, обрабатывает и выполняет исходный код программы или запроса;
* язык со слабой динамической типизацией (переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной, и, таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов);
* был разработан на основе *Perl, C, C++.*

## 1.2. Стандарты оформления кода PSR

### 1.2.1. PSR-стандарты

За время существования *PHP* было создано множество разнообразных проектов. Часть из этих проектов выросла в популярные фреймворки – готовые наборы компонентов, позволяющие быстрее создавать приложение.

Долгое время любой фреймворк представлял собой замкнутую экосистему, не совместимую с другими фреймворками. Работая в рамках одного фреймворка, разработчик не мог без адаптации привлекать компоненты, созданные для другого фреймворка. Это значительно усложняло поддержку популярных компонентов.

В 2009 году разработчики нескольких фреймворков договорились о создании сообщества *PHP Framework Interop Group (PHP-FIG)*, которое бы вырабатывало рекомендации для разработчиков.

В литературе и статьях, посвященных *PHP*, на стандарты часто ссылаются, используя аббревиатуру *PSR*, которая расшифровывается как PHP standards recommendation. Все стандарты пронумерованы, и каждый из них освещает какую-то одну проблему, часто встречающуюся при разработке больших систем на PHP, таких как фреймворк. На данный момент утверждено шесть рекомендаций:

* *PSR*-1 – основной стандарт кодирования;
* *PSR*-2 – руководство по стилю кода;
* *PSR*-3 – стандарт протоколирования;
* *PSR*-4 – стандарт автозагрузки классов;
* *PSR*-6 – стандарт кэширования;
* *PSR*-7 – стандарт *HTTP*-сообщения.

### 1.2.2. PSR-1

Первый стандарт описывает наиболее общие правила кодирования на языке *PHP*.

#### PHP-теги

В *PHP*-скриптах допускается использование только двух типов тегов – <?php ... ?> или <?= ... ?> . Альтернативные теги <% ... %>, а также <script language="php"> ... </script> исключены из *PHP* 7.

#### **Кодировка UTF-8**

Для кодирования на *PHP* допускается использовать только кодировку *UTF-8* без *BOM*-маркера (*U+FEFF*).

**Пример 1.1.** Вывод UTF-8 русской буквы А.

<?php

echo "\u{0410}";

#### Разделение объявлений и выполнения действий

В одном файле допускаются либо объявления (класса, функции, константы), либо выполнениекаких-то действий (вывод в окно браузера, изменение значений переменных)**.**

Пример 1.2. Объявление функции в отдельном файле.

<?php

function greet($\_\_name)

{

echo "<pre>";

print\_r("Hello, $\_\_name");

echo "</pre>";

}

#PSR-1 допускает условное объявление

if (!function\_exists('returnTrue')) {

function returnTrue()// объявляем функцию

{

return true;

}

}

**Пример 1.3.** Нарушение стандарта PSR-1.

<?php

#код расположен в файле psrRight.php

function greet($\_\_name)

{

echo "<pre>";

print\_r("Hello, $\_\_name");

echo "</pre>";

}

<?php

#Подключаем ранее объявленные функции

require\_once("psrRight.php");

greet("Olga");

#Нарушение PSR-1, нельзя смешивать вывод и объявления функций

function title($\_\_str)

{

echo "<h1>";

greet($\_\_str);

echo "</h1>";

}

#### Пространство имён

Невозможно использовать классы без помещения их в собственное уникальное пространство имён, при этом каждый класс должен находиться в отдельном файле.

**Пример 1.4.** Пространство имён класса. Файл Pz/Student.php.

<?php

#Каждый класс снабжается своим пространством имён

namespace Pz\Student;

class Student

{

}

#Ошибка: класс не в отдельном файле

class Officer

{

}

#### Именование классов, методов и констант

Для именования классов используется *CamelCase*-стиль: название каждого составного слова начинается с прописной буквы, пробелы не используются, например*: HelloWorld, SuperClass, Student*.

Для именования методов класса также используют *CamelCase*-стиль, однако, первая буква строчная: *superFunction(), setValue()*.

Константы классов записываются прописными буквами, при этом составные слова разделяются подчеркиванием: *HELLO\_WORLD*, *SUPER\_CONST, FLAG*.

### 1.2.3. PSR-2

Стандарт *PSR*-2 предписывает единый стиль кодирования, который позволяет создавать код в одинаковом стиле, позволяющем легко воспринимать его всем участникам команды. Первым его требованием является соблюдение *PSR*-1.

#### Отступы

Отступы в *PHP*-коде должны содержать 4 пробела, не допускается использование символа табуляции. Многие редакторы по-разному настраивают количество пробельных символов, выводящихся вместо табуляции. Поэтому использование табуляции, а тем более смешивание ее с пробелами, может приводить к совершенно нечитаемому коду. Даже если вы привыкли делать отступы при помощи табуляции, любой современный редактор можно настроить на замену табуляции пробелами.

Пример 1.5. Отступы.

<?php

#Четыре пробела перед объявлением функции

function tabber($spaces, $echo, ...$planets)

{

#По четыре пробела перед телом функции

$counter = '0';

while($counter < '5') {

++$counter; //ещё 4 пробела для тела цикла

}

echo $counter; //ошибка: отступ должен быть 4 пробела

}

#### Файлы

Во всех *PHP*-файлах следует использовать *UNIX*-переводы строк *\n*. Не допускается использование перевода строк в стиле *Windows* *\r\n* или *Mac OS* *X* *\n\r*.

В конце *PHP*-файла должна быть одна пустая строка – это позволит автоматическим системам формирования *PHP*-кода дописывать код с новой строки.

Закрывающий тег "?>" необходимо удалять из файлов, которые не содержат ничего, кроме *PHP*-кода.

#### Строки

Недопустимо использовать более одной инструкции в строке, при этом длина строки не должна превышать 80 символов без крайней необходимости. Строки обязаны быть менее 120 символов в длину и не должны содержать пробельные символы в конце.

Пример 1.6. Корректные и некорректные строки.

<?php

/\*Длинный комментарий должен быть

разбит разбит разбит разбит разбит разбит разбит разбит

на несколько строк\*/

#Ошибка: более одной инструкции на одной строке

echo "<pre>"; print\_r('Good evening'); echo "</pre>";

#### Ключевые слова

Константы *true*, *false* и *null* следует писать только в строчном регистре.

#### Пространства имён

При объявлении пространства имен после него необходимо оставлять одну пустую строку. Если производится импортирование из других пространств имен с использованием ключевого слова *use*, то пустая строка необходима также после последнего объявления *use*. Под каждое объявление нужно использовать один оператор *use*.

Пример 1.7. Отступы для пространства имён.

<?php

namespace Vendor\Hello;

use Symfony\Components\HttpFoundation\Request;

use Symfony\Components\HttpFoundation\Response;

class Hello

{

}

#### Классы

Стандарт *PSR*-2 обязывает размещать открывающую фигурную скобку класса на новой строке. Если класс содержит ключевые слова *extends* и *implements*, они также должны располагаться на строке объявления класса.

Пример 1.8. Отступы для классов.

<?php

#extends и implements на одной строке

class News extends Page implements Seo

{// открывающая скобка с новой строки

}

#Ошибка: открывающая скобка не на новой строке

class Student{

}

#### Методы

Все методы и члены классов должны предваряться модификатором доступа. При этом не следует предварять имена методов и членов класса символом подчеркивания для обозначения закрытой (*private*) или защищенной (*protected*) областей видимости.

Так же как и в случае класса, открывающая фигурная скобка должна располагаться на новой строке, а между названием класса и круглой скобкой не должно быть пробелов.

В списке аргументов не должно быть пробелов перед запятыми и должен быть один пробел после запятой. При большом количестве аргументов допускается их перенос на новую строку, при этом они должны предваряться отступом, а на каждый новый аргумент должна выделяться отдельная строка.

Когда список аргументов разбит на несколько строк, закрывающую круглую скобку и открывающую фигурную следует располагать на отдельной строке, с одним пробелом между ними.

Пример 1.9. Отступы для методов.

<?php

class Student

{

public $name;

private $id = null;

public function setParams($id, $name = "")

{

$this->id = $id;

$this->name = $name;

}

}

При наличии ключевых слов *abstract* и *final* необходимо, чтобы они предшествовали модификаторам доступа. Ключевое слово *static*, наоборот, должно располагаться за модификатором доступа.

Пример 1.10. Порядок ключевых слов abstract, final, static.

<?php

abstract class Student

{

protected static $counter;

abstract protected function data();

final public static function count()

{

self::$counter++;

}

}

#### Управляющие структуры

Для всех управляющих структур после ключевого слова необходим один пробел, после открывающей и перед закрывающей круглыми скобками, напротив, пробел не допускается.

Пример 1.11. Отступы для управляющих структур.

<?php

if (isset($\_GET['number'])) {

for ($count = 1; $count <= $\_GET['number']; ++$count) {

echo $count;

}

echo "Расчёт окончил";

}

else

{//ошибка: скобка должная быть на строке с else

echo "Не передан GET-параметр number";

}

В отличие от классов и методов открывающая круглая скобка располагается на той же строке, где и ключевое слово и отделяется от круглой скобки пробелом.

## 1.3. Компоновка программы

Рассмотрим простейший *PHP*-сценарий.

**Пример 1.12**. Простейший *PHP*-сценарий.

<?php

echo 'Hello world'; // выводит Hello world на экран

#Конец сценария (в стандарте PSR-2 сказано, когда ставить)

?>

Если запустить этот скрипт в нашей среде разработки, откроется пустая страница в браузере с текстом *Hello world.* Аналогично будет работать следующий сценарий.

Пример 1.13. Простейший *PHP*-сценарий.

<?php

printf('Hello world');

?>

Разница между *echo* и *printf*() заключается в том, что *echo* - оператор (конструкция языка), а *printf*() *–* функция.

Напишем, на первый взгляд, некорректный пример:

Пример 1.14. *HTML*-текст.

Hello

<?php

?>

world

Если запустить этот скрипт, получим такой же результат.

Итак, между тегами *<?php ?>* текст интерпретируется, как *PHP*-код и в *HTML*-документ не попадает. Если нужно что-то вывести, то пользуемся оператором *echo*, функцией *printf*() или любой другой аналогичной функцией. Текст вне тегов <*?php ?*> выводится непосредственно в браузер, то есть воспринимается как вызов оператора *echo*.

Основываясь на этом, покажем, как не надо компоновать программу.

Пример 1.15. Нежелательная компоновка кода.

<!DOCTYPE html>

<html>

<head>

<title>Нежелательная компоновка</title>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

</head>

<body>

<div>

<?php

#Кладём два слова и пробел в переменные

$firstWord = 'Hello';

$space = ' ';

$secondWord = 'world';

#Соединяем слова в переменных

$resultString = $firstWord.$space.$secondWord;

echo $resultString;// выводим результат

?>

</div>

</body>

</html>

Предпочтительнее вставлять *HTML*-код в *PHP*-код, а не наоборот, как показано в примере выше.

Пример 1.16. Предпочтительная компоновка кода.

<?php

#Кладём HTML-скрипт в переменную

$resultString = '

<!DOCTYPE html>

<html>

<head>

<title>Предпочтительная компоновка</title>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

</head>

<body>

<div>';

$firstWord = 'Hello';

$space = ' ';

$secondWord = 'world';

#Добавляем нашу строку в переменную

$resultString .= $firstWord.$space.$secondWord;

#Добавляем закрывающие тэги в результат

$resultString .= '</div></body></html>';

#выводим результат

echo $resultString;

Принцип построения кода в последнем примере таков: всё желаемое к выводу в браузер помещается в одну переменную и её содержимое печатается в конце скрипта. Так же для вывода можно использовать более удобный подход с испльзованием шаблонов, который будет рассмотрен позже.

Из примера также можно заметить, что переменные в *PHP* начинаются с $, в конце каждого смыслового выражения ставится ";", строки записываются в одинарных или двойных кавычках, строки объединяются при помощи оператора точка (".").

На примере поясним, что такое блочная компоновка кода.

Пример 1.17. Блочная компоновка.

/\*\*

\* Находит размер самого большого подмассива

\*

\* **@param** array $\_\_wordsArraysArr Массив с массивами слов.

\*

\* **@return** integer Размер самого большого подмассива

\*/

function maxWordsArrSize(array $\_\_wordsArraysArr)

{

//Максимальная длина массива

$maxArrayCount = 0;

//Вычисляем максимум поочередно сравнивая размеры массивов

foreach ($\_\_wordsArraysArr as $wordsArr) {

if (count($wordsArr) >= $maxArrayCount) {

$maxArrayCount = count($wordsArr);

}

}

//возвращаем размер самого большого массива

return $maxArrayCount;

}

Принцип блочной компоновки заключается в том, что программа визуально делится на смысловые части, предваряемые комментарием (по аналогии с абзацами в письменной речи). Комментарий описывает содержимое всего блока (со всеми внутренними конструкциями), а не первой строки блока. При сворачивании фрагментов программы комментарии должны давать представление об алгоритме работы программы. Блок программы впереди и сзади отделяется пустыми строками.

## 1.4. Базовые синтаксические конструкции

Рассмотрим базовые синтаксические конструкции, использующиеся в *PHP*.

### 1.4.1. Комментарии

В *PHP* используются комментарии трёх типов: однострочный в стиле *C++*, многострочный и комментарий в стиле оболочки *UNIX*.

Пример 1.18. Комментарии в *PHP*.

<?php

// Это однострочный комментарий

/\*

Это

многострочный

комментарий

\*/

#Комментарий в стиле UNIX

Очевидно, что, запустив этот скрипт, в браузер будет выведена пустая страница.

### 1.4.2. Унарный оператор $

Любая переменная в *PHP* должна начинаться с символа $. Необходимо обратить внимание на то, что этот символ является унарным оператором, который возвращает значение. Рассмотрим, особенность использования этого оператора на примере.

Пример 1.19. Использование оператора $.

<?php

#Присваиваем значения переменным

$tree = "apple";

$apple = "seed";

#Выводим значения переменных

// выведет apple

echo $tree;

// выведет seed

echo $apple;

// выведет seed

echo $$tree;

В выражении $$*tree* на место $*tree* подставляется строка *apple*, которая в последствии сама используется как название переменной $*apple*.

### 1.4.2. Типы переменных

Рассмотрим типы переменных, использующихся в *PHP*. Забегая вперед заметим, что тип переменной можно узнать при помощи вызова функции *gettype*($*variable*), которая примет значение, равное имени типа в строковом представлении.

#### boolean (логический тип)

Логическая переменная может содержать одно из двух значений: *false* (ложь) или *true* (истина). Вообще, любое ненулевое число (и непустая строка), а также ключевое слово *true* символизирует истину, тогда как 0, пустая строка и слово *false* – ложь. Таким образом, любое ненулевое выражение (в частности, значение переменной) рассматривается в логическом контексте как истина. Вы можете пользоваться константами *false* и *true* в зависимости от логики программы.

При выполнении арифметических операций над логической переменной она преобразуется в обычную, числовую переменную. То есть, *false* рассматривается как 0, а *true* – как 1.

#### *integer* (целое число)

Целое число со знаком, размер которого зависит от разрядности *PHP*. В 32-битном варианте целое число может принимать значение от –2 147 483 648 до 2 147 483 647. В 64-битном от –9 223 372 036 854 775 807 до 9 223 372 036 854 775 807. Выяснить, максимальное значение для целого числа в *PHP* можно, обратившись к предопределенной константе *PHP\_INT\_MAX*.

При выходе за диапазон целое число автоматически преобразуется в число типа *double*.

#### *double* (вещественный тип)

Числа с плавающей точкой имеют две формы записи. Обычная форма совпадает с принятой в арифметике, например, 346.1256. Экспоненциальная форма позволяет представить числа в виде произведения мантиссы 3.461256 и соответствующей степени числа 10. Для цифр меньше нуля степень числа 10 является отрицательной.

В компьютерных программах нет возможности использовать символы верхнего регистра, поэтому конструкцию ×10 записывают в виде символа *e*, после которого указывается значение степени.

Пример 1.20. Запись вещественного типа.

<?php

// Обычная форма

$firstNumber = 25.12;

// Экспоненциальная форма 25.12

$secondNumber = 2.512e+1;

При выходе за допустимый диапазон вместо числа выводится константа *INF*, символизирующая бесконечность. Любые операции с таким числом опять возвращают *INF*.

Кроме константы *INF* в *PHP* реализована константа *NAN*, которая обозначает недопустимое число (например, квадратный корень из -1).

#### *string* (строка текста)

Строка практически любой длины. В отличие от *C*, строки могут содержать в себе также и нулевые символы, что никак не повлияет на программу. Иными словами, строки можно использовать для хранения бинарных данных. Длина строки ограничена 2 Гбайт. Однако этот предел редко достигается, т. к. каждый *PHP*-скрипт ограничен в объеме потребляемой памяти директивой *limit*, который по умолчанию составляет

128 Мбайт. Так что вполне возможно прочитать в одну строку содержимое целого файла размером в несколько мегабайт (что часто и делается). Строка легко может быть обработана при помощи стандартных функций, допустимо также непосредственное обращение к любому ее символу.

Строка в *PHP* помещается в одинарные или двойные кавычки. Разница между ними состоит в том, что переменные и управляющие последовательности для специальных символов, заключенных в одинарные кавычки, не обрабатываются.

Пример 1.21. Одинарные и двойные кавычки.

<?php

echo 'Это просто текст';

echo "И это тоже просто текст";

$count = 20;

#Выведет - В этом предложении 20 букв

echo "В этом предложении $count букв";

$count = 10;

#Так и выведет - А в этом $count букв

echo 'А в этом $count букв';

Чтобы использовать одинарную кавычку внутри строки, необходимо проэкранировать ее обратным слешем (\). При необходимости написать сам обратный слеш, продублируйте его (\\). Все остальные случаи применения обратного слеша будут интерпретированы как обычные символы.

Если строка заключена в двойные кавычки, *PHP* распознает следующие управляющие последовательности специальных символов: *\n*, *\r*, *\t*, *\v* и так далее.

#### array (массив)

*PHP* позволяет хранить наборы элементов, при этом для доступа к ним могут использоваться как числовые индексы (тогда массив называется список), так и строковые (тогда массив называется ассоциативным). Рассмотрим базовый синтаксис обращения к элементу массива:

Пример 1.22. Обращение к элементу массива.

<?php

$phonesArr = array('iPhone', 'Samsung Galaxy', 'Nokia');

echo $phones[1];

Более подробно синтаксис и функции работы с массивами будут рассмтрены далее в главе 3.

#### object (объект)

Объект представляет собой переменную, экземпляр которой создается по специальному шаблону, называемому классом. Более подробно объекты будут рассмотрены в главе 8.

#### resource (ресурс)

Ресурс представляет специальную тип данных, который используется для хранения связи с внешним ресурсом. В качестве внешнего ресурса могут использоваться, например, файлы или подключения к базам данных. Ресурсы создаются и используются специальными функциями. Далее мы подробнее рассмотрим работу с файлами и подключения к базе данных.

#### null

Значение null указывает, что значение переменной не определено. Использование данного значения полезно в тех случаях, когда мы хотим указать, что переменная не имеет значения. Например, если мы просто определим переменную без ее инициализации, и затем попробуем ее использовать, то нам интерпретатор выдаст сообщение, что переменная не установлена.

### 1.4.3. Действия с переменными

#### Присвоение значения

В *PHP* имеется возможность присвоить некоторой переменной значение другой переменной (или значение, возвращенное функцией), ссылку на другую переменную, либо же константное выражение. Если переменная первый раз встречается в программе, происходит ее инициализация. Исключение составляют объекты, которые инициализируются явно оператором *new*. Кроме того, при присваивании старое содержимое и, что самое важное, тип переменной теряются, и она становится абсолютно точной копией своего "родителя". То есть, если массиву присвоить число, это сработает, однако весь массив при этом будет утерян.

#### Проверка существования

Можно проверить, существует ли (то есть инициализирована ли) указанная переменная. Выполняется это при помощи встроенной в *PHP* конструкции *isset*().

Пример 1.23. Проверка существования переменной.

<?php

if (isset($myVar)) {

echo 'Переменная $myVar существует';

}

else {

echo 'Переменная $myVar не инициализирована';

}

Невозможно использовать неинициализированную переменную в программе – иначе это породит предупреждение со стороны интерпретатора. Категорически не рекомендуется отключать предупреждения, но в случае если это необходимо, то можно использовать оператор отключения ошибок.

#### Уничтожение

Уничтожение переменной реализуется оператором *unset*(). После этого переменная удаляется из внутренних таблиц интерпретатора, то есть программа начинает выполняться так, как будто переменная еще не была инициализирована.

Пример 1.24. Уничтожение переменной.

<?php

// инициализируем переменную

$myVar = 'Hello';

if (isset($myVar)) {

echo 'Переменная $myVar существует';// ok

}

else {

echo 'Переменная $myVar не инициализирована';

}

// уничтожаем переменную

unset($myVar);

if (isset($myVar)) {

echo 'Переменная $myVar существует'; }

else {

// Так как переменной теперь не существует

echo 'Переменная $myVar не инициализирована';

}

Впрочем, применение *unset*() для работы с обычными переменными редко бывает целесообразно. Куда как полезнее использовать его для удаления элемента в ассоциативном массиве. Например, если в массиве *$students* нужно удалить элемент с ключом *yegor*, это можно сделать следующим образом:

unset($students["yegor"]);

Теперь элемент с ключом *"yegor"* был удален, и при последующем просмотре массива, он не будет обнаружен.

#### Определение типа переменной

Существуют несколько стандартных функций, которые занимаются определением типа переменных и часто включаются в условные операторы:

* *is\_int*();
* *is\_double*();
* *is\_bool*();
* *is\_string*();
* *is\_infinite*();
* *is\_nan*();
* *is\_array*().

Не трудно догадаться, какие типы проверяет каждая из функций. В случае если выполняется, возвращают *true*.

Функция *gettype*() возвращает тип переменной. Если тип не является встроенным в *PHP*, то *gettype*()возвращает *unknown type*.

#### Установка типа переменной

Существует функция, которая пытается привести тип указанной переменной к одному из стандартных: *settype*(*$var*, *$type*)*.* Функция пытается привести тип переменной *$var* к типу *$type* (*$type* – одна из строк, возвращаемых *gettype*(), кроме *boolean*). Если это сделать не удалось (например, в *$var* "нечисловая" строка, а вызывается *settype*(*$var*, "*integer*")), возвращает *false*.

Помимо функции *settype*() существуют и более специализированные функции преобразования, например: *floatval*(*$var*). Функция преобразует переменную *$var* к вещественному числу. Для нее существует псевдоним *doubleval*(). Обе функции совершенно эквивалентны.

Функция *strval*(*$var*) преобразует переменную *$var* в строку.

Функция *intval*(*$var*[, *$base*]) преобразует переменную *$var* в целочисленную переменную. По умолчанию числа приводятся к привычному десятичному формату. Однако при помощи необязательного параметра *$base* можно получить результат в другой системе счисления.

Кроме указанных функций, в отношении переменных *PHP* действует *C*-синтаксис приведения типа, с указанием типа в круглых скобках перед переменной. Например (*int*), (*string*), (*boolean*).

#### Оператор присваивания

В *PHP* оператором присваивания является знак равенства (=):

$имя\_переменной = значение;

Как видите, разработчики *PHP* пошли по линии языка *C* в вопросе операторов присваивания (и проверки равенства, которая обозначается ==), чем, вероятно, привнесли свой вклад в размножение многочисленных ошибок. Например, если в *C* пишется

if (a = b) { ... }

вместо

if (a == b) { ... }

(упуская один символ равенства), то компилятор выдаст, по крайней мере, предупреждение. Но в *PHP* иначе. Так что, запустив следующий код:

Пример 1.25. Неверная проверка на равенство.

<?php

$a = 1;

$b = 999;

if ($a = $b) {

echo 'Числа равны';

}

else {

echo 'Числа не равны';

}

Программа выведет: "Числа равны".

### 1.4.4. Ссылочные переменные

Хотя в *PHP* нет такого понятия, как указатель, все же можно создавать ссылки на другие переменные. Существуют три разновидности ссылок: жесткие, символические и ссылки на объекты (первые часто называют просто ссылками).

#### Жёсткие ссылки

Жесткая ссылка представляет собой просто переменную, которая является синонимом другой переменной. Чтобы создать жесткую ссылку, нужно использовать оператор (=&). Например:

Пример 1.26. Использование жёстких ссылок.

<?php

$a = 'yegor';

// теперь $b то же самое, что и $a

$b =& $a;

// на самом деле $a = 'pomidor'

$b = 'pomidor';

// соответственно, будет выведено 'pomidor'

echo $a;

Ссылаться можно не только на переменные, но и на элементы массива (этим жесткие ссылки выгодно отличаются от символических). Элемент массива при этом может не существовать (тогда будет создан элемент с переданным ключом и *null* значением).

#### Символические ссылки

Символическая ссылка – это всего лишь строковая переменная, хранящая имя другой переменной. Чтобы добраться до значения переменной, на которую указывает символическая ссылка, необходимо применить оператор разыменования – дополнительный знак $ перед именем ссылки.

Символическая ссылка уже была рассмотрена в примере 1.19.

Заметим, что для использования обычной строковой переменной в качестве ссылки нужно перед ней поставить еще один символ "$". Это говорит интерпретатору, что надо взять не значение самой переменной $*tree*, а значение переменной, имя которой хранится в переменной $*tree*.

### 1.4.5. Константы

Встречаются случаи, когда значения некоторых величин не меняются в течение работы программы. Это могут быть математические константы, пути к файлам, разнообразные пароли и так далее. Как раз для этих целей в *PHP* предусмотрена такая конструкция, как константа.

Константа отличается от переменной тем, что, во-первых, ей нигде в программе нельзя присвоить значение больше одного раза, а во-вторых, ее имя не предваряется знаком "$", как это делается для переменных. Например:

Пример 1.27. Использование константы.

<?php

// определяем константу

define("ANSWER\_ALL\_QUESTIONS", 42);

// так и выведет в браузер

echo 'Ответ на все вопросы ANSWER\_ALL\_QUESTIONS';

// ок, 'Ответ на все вопросы 42'

echo 'Ответ на все вопросы '.ANSWER\_ALL\_QUESTIONS;

Как видно из примера, у констант, из-за того, что не нужно писать знак доллара перед именем, есть особенность: мы уже не можем использовать имя константы непосредственно в текстовой строке. Имя константы зависит от регистра.

#### Предопределённые константы

Константы бывают двух типов: одни – предопределенные (то есть устанавливаемые самим интерпретатором), а другие определяются программистом.

Рассмотрим несколько предопределенных констант:

* *\_\_FILE\_\_* (хранит имя файла, в котором расположен запущенный в настоящий момент код);
* \_\_*LINE*\_\_ (содержит текущий номер строки, которую обрабатывает в текущий момент интерпретатор);
* \_\_*FUNCTION*\_\_ (имя текущей функции);
* \_\_*CLASS*\_\_ (имя текущего класса);
* *PHP\_VERSION* (версия интерпретатора);
* *PHP\_OS* (имя операционной системы);
* *PHP\_EOL* (символ конца строки, используемый на текущей платформе);
* *true* и *false*;
* *null*.

#### Определение констант

void define(string $name, string $value, bool $case\_sen = true);

Определяет новую константу с именем, переданным в $*name*, и значением $*value*. Если необязательный параметр $*case\_sen* равен *true*, то в дальнейшем в программе регистр букв константы учитывается, в противном случае – игнорируется (по умолчанию, как мы видим, регистр учитывается). Созданная константа не может быть уничтожена или переопределена.

Использование функции было рассмотрено в примере 1.27.

Обратите внимание на кавычки, которыми должно быть обрамлено имя константы при ее определении. А также на то, что нельзя дважды определять константу с одним и тем же именем – это породит ошибку во время выполнения программы.

### 1.4.6. Выражения

Программы состоят из выражений, имеющих определенные значения. С другой стороны, если что-то имеет значение, то это выражение.

#### Логические выражения

Абсолютно любое выражение может рассматриваться как логическое в "логическом" же контексте (например, как условие для конструкции *if-else*). Ведь в качестве истины может выступать любое ненулевое число, непустая строка и т. д., а под ложью подразумевается все остальное.

Эти выражения чаще всего возникают при применении операторов ">", "<" и "==" (равно), "||" (логическое ИЛИ), "&&" (логическое И), "!" (логическое НЕ) и других. Например:

Пример 1.28. Логические выражения.

<?php

// очевидно, что ложь не истина

$a = true != false;

// истина, 18 больше 16

$b = 18 > 16;

// истинно, если $a и $b истинны

$c = $a && $b;

#### Строковые выражения

Строки в *PHP* – одни из основных объектов. Как мы уже говорили, они могут содержать текст вместе с символами форматирования или даже бинарные данные. Определение строки в кавычках или апострофах может начинаться на одной строке, а завершаться – на другой.

Разница между строками, заключёнными в кавычки и апострофы, рассмотрена в примере 1.21.

### 1.4.7. Операции

Рассмотрим операции в *PHP*, заменяя выражения буквами *a* и *b*.

Арифметические операции

Перечислим их:

* *a* + *b* – сложение;
* *a* – *b* – вычитание;
* *a* \* *b* – умножение;
* *a* / *b* – деление a на b;
* *a* % *b* – остаток от деление a на b;
* *a* \*\* *b* – возведение a в степень b.

Операция деления "/" возвращает целое число (т. е. результат деления нацело), если оба выражения *a* и *b* – целого типа (или же строки, выглядящие как целые числа), в противном случае результат будет дробным. Операция вычисления остатка от деления "%" работает только с целыми числами, так что применение ее к дробным числам может привести, мягко говоря, к нежелательному результату. Оператор возведения в степень был введен, начиная с версии *PHP* 5.6, и допускает, в том числе, дробный аргумент степени. Это позволяет не только возводить числа в целую степень, но и, например, извлекать квадратный корень.

Строковые операции

К ним относятся:

* *a.b* – слияние строк;
* *a*[*n*] – символ строки в позиции *n*.

Собственно, других строковых операций и нет – все остальное, что можно сделать со строками в PHP, выполняют стандартные функции (вроде *strlen*(), *substr*()).

Операции присваивания

Основным из этой группы операций является рассмотренный ранее оператор присваивания "=".

Помимо этого основного оператора существует еще множество комбинированных – по одному на каждую арифметическую, строковую и битовую операцию. Например:

* *a* +=*b* , *a* −= *b*;
* *a* \*= *b*, *a* /= *b*;
* *a* %= *b*;
* *a* .= *b*;
* *a* &= *b*, *a* |= *b*.

Операции инкремента и декремента

* *a*++ − увеличение на 1;
* *b*−− − уменьшение на 1.

Как и в языке *C*, эти операторы увеличивают или уменьшают значение переменной, а в выражении возвращают значение переменной до изменения.

Существуют также парные рассмотренным операторы, которые указываются до, а не после имени переменной. Соответственно, и возвращают они значение переменной уже после изменения.

Пример 1.29. Инкремент и декремент.

<?php

$num = 0;

// $num = 1

$num++;

// будет выведено 1, после чего $num увеличится

echo $num++;

//ок, $num = 2

$num;

// будет выведено 3

echo ++$num;

//$num = 2

--$num;

Битовые операции

Эти операции предназначены для работы (установки/снятия/проверки) групп битов в целой переменной. Биты целого числа – это не что иное, как отдельные разряды того же самого числа, записанного в двоичной системе счисления. Например, в двоичной системе число 12 будет выглядеть как 1100, а 2 – как 10, так что выражение 12|2 вернет нам число 14 (1110 в двоичной записи). Если переменная не целая, то она вначале округляется, а уж затем к ней применяются перечисленные ниже операторы.

* *a* & *b* (число с установленными битами, которые выставлены и в *a*, и в *b* одновременно);
* *a* | *b* (число с установленными битами, которые выставлены либо в *a*, либо в *b*, либо одновременно);
* *a* ^ *b* (число с установленными битами, которые выставлены либо в *a*, либо в *b*);
* ~ *a* (результат, у которого на месте единиц в *a* стоят нули, и наоборот);
* *a* << *b* (число, полученное поразрядным сдвигом *a* на *b* битов влево);
* *a* >> *b* (наоборот).

Битовые операции довольно интенсивно используются в директивах и функциях *PHP*, т. к. они позволяют зашифровать в одном целом числе несколько значений. Как правило, битовые значения задаются при помощи констант, которые затем можно использовать в скрипте.

Операции сравнения

Эти операции независимо от типов своих аргументов всегда возвращают *true* или *false*.

Операции сравнения позволяют сравнивать два значения между собой и, если условие выполнено, возвращают *true*, в противном случае – *false*.

* *a* == *b* − истина, если *a* равно *b*;
* *a* != *b* − истина, если *a* не равно *b*;
* *a* === *b* − истина, если *a* эквивалентно *b*;
* *a* !== *b* − истина, если *a* не эквивалентно *b*;
* *a* < *b* − истина, если *a* меньше *b*;
* *a* > *b* − истина, если *a* больше *b*;
* *a <= b* − истина, если *a* меньше либо равно *b*;
* *a >= b* − истина, если *a* больше либо равно *b*;
* *a <=> b* − возвращает –1, если *a* меньше *b* , 0, если *a* равно *b* и 1, если *a* больше *b*.

Логические операции

Эти операции предназначены исключительно для работы с логическими выражениями и также возвращают *false* или *true*:

* !*a;*
* *a* && *b;*
* *a* || *b.*

Смысл этих операций очевиден из математической логики.

Следует заметить, что вычисление логических выражений, содержащих такие операции, идет всегда слева направо, при этом если результат уже очевиден, то вычисления обрываются, даже если в выражении присутствуют вызовы функций.

Для операторов "&&" и "| |" существуют аналогичные операторы *and* и *or*, которые отличаются только более низким приоритетом, что позволяет не заключать в скобки выражения справа и слева от оператора.

## 1.5. Операторы изменения хода выполнения программы

### 1.5.1. Условные операторы

#### if – elseif – else

Конструкция *if* является одной из наиболее важных во многих языках программирования, в том числе и *PHP*. Она предоставляет возможность условного выполнения фрагментов кода. Структура *if* реализована в *PHP* по аналогии с языком *C*:

if (логическое\_выражение) {

// инструкция\_1;

}

else {

// инструкция\_2;

}

Конструкция *elseif*, есть сочетание *if* и *else*. Аналогично *else*, она расширяет оператор *if* для выполнения различных выражений в случае, когда условие начального оператора *if* эквивалентно *FALSE*. Однако, в отличие от *else*, выполнение альтернативного выражения произойдет только тогда, когда условие оператора *elseif* будет являться равным *TRUE*.

Пример 1.30. Использование *elseif*.

<?php

$a = 5;

if ($a > 10) {

// не выполнится;

}

elseif ($a > 5) {

// не выполнится;

}

elseif ($a > 0) {

// выполнится;

}

else {

// не выполнится;

}

#### Тернарный оператор

*РНР* так же, как и любой *С*-подобный язык, предоставляет условный оператор, который возвращает *y*, в случае если x принимает значение *true*, и *z* в случае, если *x* принимает значение *false*:

x ? y : z

При написании кода целесообразно использовать тернарный оператор, в случае, если при ветвлении в теле if и else выполняется похожая по смыслу операция. Рассмотрим использование тернарного оператора на примере 1.31.

Пример 1.31. Использование тернарного оператора.

<?php

$num = -42;

//код без тернарного оператора менее читабельный:

if ($num >= 0) {

$num = $num;

}

else {

$num = -$num;

}

//код с тернарным оператором:

$num = ($num >= 0) ? $num : -$num;

// теперь $num будет выведено по модулю

echo $num;

#### switch-case

Оператор *switch* похож на ряд операторов *if* с одинаковым условием. Во многих случаях вам может понадобиться сравнивать одну и ту же переменную (или выражение) с множеством различных значений и выполнять различные участки кода в зависимости от того, какое значение принимает эта переменная (или выражение). Именно в этом случае удобен оператор *switch*:

switch (выражение) {

case значение1: команды1; [break;]

[default: команды\_по\_умолчанию; [break;]]

case значение2: команды2; [break;]

. . .

case значениеN: командыN;

}

Этот оператор работает по следующему принципу: вычисляет значение выражения (пусть оно равно, например, "*V*"), а затем пытается найти строку, начинающуюся с *case* "*V*". Если такая строка обнаружена, выполняются команды, расположенные сразу после него. Если же найти такую строку не удалось, выполняются команды после *default*.

Обратите внимание на операторы *break*, добавленные после каждой строки команд, кроме последней, для которой можно было бы тоже указать *break*, что не имело бы смысла. Если бы не они, то сработали бы не только команды, относящиеся к *V*, но и все нижележащие.

Пример 1.32. Использование *switch-case.*

<?php

// наш логин

$name = 'Городецкий';

switch ($name) {

case 'Гессер':

echo 'Приветствую Борис Игнатьевич';

break;

case 'Городецкий': // будет выведено в нашем случае

echo 'Приветствую Антон';

break;

case 'Ольга': // сработает на любое из трёх имен

case 'Света':

case 'Павел':

echo 'Приветствую, Светлый';

break;

default: // если не было совпадений

echo 'Уходи, Тёмный';

break;

}

### 1.5.2. Циклы

#### while()

Циклы *while* являются простейшим видом циклов в *PHP*. Они ведут себя так же, как и в языке *C*.

Простейшей формой цикла *while* является следующее выражение:

while (логическое\_выражение) {

инструкция;

}

Смысл выражения *while* состоит в том, что оно указывает *PHP* выполнять вложенные выражения повторно до тех пор, пока выражение в самом *while* является *TRUE*. Значение логического выражения проверяется каждый раз перед началом цикла, поэтому даже если значение выражения изменится в процессе выполнения вложенных выражений, шаг цикла все равно будет выполнен до конца. В том случае, если выражение *while* равно *FALSE* с самого начала, вложенные выражения ни разу не будут выполнены.

Так же, как и с оператором *if*, вы можете группировать несколько выражений внутри одного цикла *while*, заключая эти выражения между фигурными скобками.

Пример 1.33. Использование *while*().

<?php

$number = 7; #Посчитаем факториал 7

$result = 1; #Сюда будем записывать результат

// Вычисляем факториал

while ($number > 0) {

$result \*= $number;

// уменьшаем $number на 1

--$number;

}

// выводим результат

echo $result;

Цикл *do-while* очень похож на цикл *while*:

do {

команды;

} while (логическое\_выражение);

Истинность выражения в таком случае проверяется в конце итерации, а не в начале. Главное отличие от обычного цикла *while* в том, что первая итерация цикла *do-while* гарантированно выполнится (истинность выражения проверяется в конце итерации), тогда как она может не выполниться в обычном цикле *while*.

#### for()

Цикл *for* в *PHP* ведет себя подобно языку *C*. Синтаксис цикла *for* следующий:

for (expr1; expr2; expr3) {

statement;

}

Первое выражение (*expr1*) всегда вычисляется (выполняется) только один раз в начале цикла. В начале каждой итерации оценивается выражение *expr2*. Если оно принимает значение *TRUE*, то цикл продолжается, и выполняются вложенные операторы. Если оно принимает значение *FALSE*, выполнение цикла заканчивается. В конце каждой итерации выражение *expr3* вычисляется (выполняется).

Каждое из выражений может быть пустым или содержать несколько выражений, разделенных запятыми. В *expr2* все выражения, разделенные запятыми, вычисляются, но результат берется из последнего. Если выражение *expr2* отсутствует, это означает, что цикл будет выполняться бесконечно. (*PHP* неявно воспринимает это значение как *TRUE*, так же, как в языке *C*).

Пример 1.34. Использование цикла *for*().

#Посчитаем факториал из прошлого примера

$number = 7;

$result = 1;

// Вычисляем факториал

for ($number; $number > 0; --$number) {

$result \*= $number;

}

// выводим результат

echo $result;

#### foreach()

Конструкция *foreach*() предоставляет простой способ перебора массивов. *foreach*() работает только с массивами и объектами и будет генерировать ошибку при попытке использования с переменными других типов или неинициализированными переменными. Существует два вида синтаксиса:

foreach (array\_expression as $value) {

//statement;

}

foreach (array\_expression as $key => $value) {

//statement;

}

Первый вариант конструкции *foreach* позволяет поочередно обращаться к элементам списков, задаваемых с помощью *array\_expression*. На каждой итерации значение текущего элемента присваивается переменной *$value.*

Пример 1.35. Использование *foreach*().

<?php

#Массив студентов

$studentsArr = array('1' => 'Гессер', '4217' => 'Ольга', '9834' => 'Антон', '8524' => 'Света');

#Будем выводить имя студента и его id

foreach ($studentsArr as $id => $name) {

echo "$name имеет id = $id ";

}

Второй вариант конструкции *foreach* дополнительно позволяет получить ключ текущего элемента и присвоить его в переменную *$key*.

Для того, чтобы изменять значения элементов массива внутри цикла, переменной *$value* должен предшествовать знак "*&*". В этом случае значение будет присвоено по ссылке:

Пример 1.36. Использование ссылок в *foreach*().

<?php

$numberArr = array(1, 2, 3, 4);

foreach ($numberArr as &$value) {

$value = $value \* 2;

}

#Массив $numberArr сейчас выглядит так: array(2, 4, 6, 8)

#### Операторы break, continue, return, goto

*break* прерывает выполнение текущей структуры *for*, *foreach*, *while* или *switch*.

*break* принимает необязательный числовой аргумент, который сообщает ему выполнение какого количества вложенных структур необходимо прервать. Значение по умолчанию 1, только ближайшая структура будет прервана.

*continue* используется внутри циклических структур для пропуска оставшейся части текущей итерации цикла и, при соблюдении условий, начала следующей итерации.

*continue* принимает необязательный числовой аргумент, который указывает на скольких уровнях вложенных циклов будет пропущена оставшаяся часть итерации. Значением по умолчанию является 1, при которой пропускается оставшаяся часть текущего цикла.

*return* возвращает управление программой модулю, из которого была вызвана функция. Выполнение программы продолжается с инструкции, следующей за местом вызова.

Оператор *goto* используется для перехода в другую часть программы. Место, куда необходимо перейти указывается с помощью метки, за которой ставится двоеточие, после оператора *goto* указывается желаемая метка для перехода. Отметим, что использование этого оператора явлется нежелательным, так как код становится запутанным.

### 1.5.3. Включение кода

#### include

Выражение *include* включает и выполняет указанный файл.

Файлы включаются исходя из пути указанного файла, или, если путь не указан, используется путь, указанный в директиве *include\_path*. Если файл не найден в *include\_path*, *include* попытается проверить директорию, в которой находится текущий включающий скрипт и текущую рабочую директорию перед тем, как выдать ошибку. Конструкция *include* выдаст предупреждение, если не сможет найти файл.

Если путь указан – абсолютный (начинающийся с буквы диска или с \ в *Windows* или с / в *Unix/Linux* системах) или относительно текущей директории (начинающийся с "." или "..") – *include\_path* будет проигнорирован вообще. Например, если имя файла начинается с ".."/, парсер будет искать в родительской директории запрошенный файл.

#### require

*require* аналогично *include*, за исключением того, что при ошибке он также выдаст фатальную ошибку уровня E\_COMPILE\_ERROR. Другими словами, он остановит выполнение скрипта, тогда как *include* только выдал бы предупреждение *E\_WARNING*, которое позволило бы скрипту продолжить выполнение.

#### include\_once и require\_once

Выражения включают и выполняют указанный файл во время выполнения скрипта. Их поведение идентично выражениям *include* и *require*, с той лишь разницей, что если код из файла уже один раз был включен, он не будет включен и выполнен повторно и вернёт *TRUE*. Как видно из имени, он включит файл только один раз.

Ими можно пользоваться в тех случаях, когда один и тот же файл может быть включен и выполнен более одного раза во время выполнения скрипта, в данном случае это поможет избежать проблем с переопределением функций, переменных и т.д.

# Глава 2. Использование функций в PHP

## 2.1. Функции в PHP

### 2.1.1. Особенности синтаксиса описания функций PHP

Синтаксис описания функций *PHP* имеет следующие особенности:

* возможно использовать параметры по умолчанию;
* каждая функция имеет собственную область видимости переменных;
* существует инструкция *return*, позволяющая вернуть результат вычисления из функции в вызывающую программу;
* тип возвращаемого значения может быть любым;
* при описании методов классов для параметров и возвращаемого значения функций возможно указание их типа (как в *C*++) с принудительной проверкой при вызове;
* допускается создание анонимных функций.

### 2.1.2. Общий синтаксис определения функции

Упрощенно синтаксис функции можно представить следующим образом:

Пример 2.1. Пример синтаксиса функции.

// Здесь арг – имя аргумента функции, зн – значение по

// умолчанию.

function имяФункции(арг1[=зн1], ..., аргN[=знN])

{

операторы\_тела\_функции;

}

Имя функции не зависит от регистра, но должно быть уникально среди имен уже объявленных функций. Это означает, что, во-первых, имена *MyFunction*, *Myfunction* и даже *MyFuNcTiOn* будут считаться одинаковыми, и, во-вторых, нельзя переопределить уже определенную функцию, но зато разрешено давать функциям такие же имена, как и переменным (без знака $ в начале). Список аргументов состоит из нескольких перечисленных через запятую переменных, каждую из которых мы должны будем задать при вызове функции.

Рассмотрим пример функции. Во многих скриптах, использующих формы с полями <*select*> (выпадающий список), элементы списка берутся из какого-нибудь внешнего источника (файла, базы данных и т. д.). Пусть они записаны в некоторый массив и необходимо его представить в виде элементов выпадающего списка.

Пример 2.2. Пример функции и ее использования.

/\*\*

\* собирает тег select с внутренними option по массиву с фамилиями

\* **@param** array $\_\_peopleArr - массив с данными вида id => surname

\* **@param** int $\_\_selected - номер выбранного элемента

\* **@return** string - html-код тега select

\*/

function selectItems($\_\_peopleArr, $\_\_selected = 0)

{

$html = "<select>";

//собираем группу option

foreach ($\_\_peopleArr as $id => $surname) {

//проверяем, нужно ли выбрать текущий option

$style = ($id === $\_\_selected) ? " selected" : "";

$html .= "<option$style value='$id'>$surname</option>\n";

}

return $html."</select>";

}

### 2.1.3. Возврат значений, инструкция return

Значения возвращаются при помощи необязательного оператора return. Возврат приводит к завершению выполнения функции и передаче управления обратно к той строке кода, в которой данная функция была вызвана. Синтаксис [*return*](http://php.net/manual/ru/function.return.php) абсолютно тот же, как и в языке *C*, за исключением одной очень важной детали. Если в *C* функции очень редко возвращают большие объекты, а массивы они не могут возвратить вовсе, то в *PHP* можно использовать [*return*](http://php.net/manual/ru/function.return.php) абсолютно для любых объектов, причем без заметной потери быстродействия. Сразу несколько значений функции, разумеется, возвратить не могут. Однако если это все же очень нужно, то можно вернуть ассоциативный массив или же список.

Пример 2.3. Пример функции, возвращающей квадрат аргумента.

function mySqr($\_\_n)

{

return $\_\_n \* $\_\_n;

}

$value = mySqr(4);

В этом примере использована конструкция *list*() для распределения значений массива по переменным.

Пример 2.4. Возврат массива функцией.

function silly()

{

return [1, 2, 3];

}

Для того, чтобы функция возвращала результат по ссылке, необходимо использовать оператор & и при описании функции, и при присвоении переменной возвращаемого значения:

Пример 2.5. Возврат ссылки.

function &referenceReturn()

{

return $someRef;

}

$newref = & referenceReturn();

### 2.1.4. Объявление и вызов функции

Функцию можно создавать не только в определенном месте программы, но и прямо среди других инструкций. Фактически объявление функции само является инструкцией. Например, вполне можно было бы поместить функцию *selectItems*(), рассмотренную в примере 2.2, прямо в середину кода:

**Пример 2.6.** Пример размещения функции в коде.

echo "Программа...";

function selectItems($\_\_items, $\_\_selected = 0) {

// ... тело функции ...

}

echo "Программа продолжается!";

При таком подходе транслятор, дойдя до определения функции, просто проверит его корректность и оттранслирует во внутреннее представление, но не будет генерировать код для выполнения. Вместо этого он сразу переключится на следующие за телом функции команды. Только потом, при вызове функции, интерпретатор начнет исполнять ее команды. Поскольку фазы трансляции и исполнения в *PHP* разделены, можно применять вызовы функции еще до того, как она была описана. Например, если переставить определение функции из начала файла в его конец**,** то сценарий по-прежнему будет функционировать. Определение функции ниже ее вызова, конечно же, работает только в том случае, если в момент вызова функции ее код будет уже оттранслирован. Например, вызов и описание функции происходят в одном и том же файле. В частности, если попробовать поместить функцию во внешний файл, который затем включить по инструкции *require\_once*, вызвать функцию получится лишь после данной инструкции.

### **2.1.5. Встроенные и анонимные функции**

В самом *PHP* содержится достаточно большое количество встроенных функций и языковых конструкций. Также есть функции, которые требуют, чтобы *PHP* был собран со определенными расширениями, в противном случае будут генерироваться фатальные ошибки, вызванные использованием неизвестной функции. Например, для того чтобы использовать функции для работы с изображениями, например, *imagecreatetruecolor*(), необходимо собрать *PHP* с поддержкой *GD*. Тем не менее, есть много встроенных функций, которые доступны всегда: например, функции обработки строк и функции для работы с переменными. Вызвав *phpinfo*() или *get\_loaded\_extensions*(), можно узнать, поддержка каких модулей есть в используемом *PHP*. Также следует учесть, что поддержка некоторых дополнительных расширений включена по умолчанию, и что сама документация к *PHP* разбита по расширениям.

Анонимные функции, также известные как замыкания (*closures*), позволяют создавать функции, не имеющие определенных имен. Они наиболее полезны в качестве значений *callback*-параметров, но также могут иметь и множество других применений. Анонимные функции реализуются с использованием класса *Closure*.

Пример 2.7. Пример анонимной функции.

echo preg\_replace\_callback(

'~-([a-z])~',

function ($\_\_match)

{

return strtoupper($\_\_match[1]);

},

'hello-world'

);

Замыкания также могут быть использованы в качестве значений переменных. *PHP* автоматически преобразует такие выражения в экземпляры внутреннего класса *Closure*. Присвоение замыкания переменной использует тот же синтаксис, что и для любого другого присвоения, включая завершающую точку с запятой:

Пример 2.8. Пример присвоения анонимной функции переменной.

$greet = function($\_\_name)

{

printf("Привет, %s\r\n", $\_\_name);

};

$greet('Мир');

Замыкания могут также наследовать переменные из родительской области видимости. Любая подобная переменная должна быть объявлена в конструкции *use*.

Пример 2.9. **Наследование переменных из родительской области видимости.**

$message = 'привет';

// Наследуем $message

$example = function () use ($\_\_message)

{

var\_dump($\_\_message);

};

$example();

Наследование переменных из родительской области видимости не то же самое, что использование глобальных переменных. Глобальные переменные существуют в глобальной области видимости, которая не меняется, вне зависимости от того, какая функция выполняется в данный момент. Родительская область видимости – это функция, в которой было объявлено замыкание (не обязательно та же самая, из которой оно было вызвано).

### 2.1.6. Функции с переменным количеством параметров

Функция может иметь несколько параметров, заданных по умолчанию. Они перечисляются справа налево, и их всегда фиксированное количество. Однако такая схема может устроить нас не всегда. Например, если необходимо написать функцию наподобие *echo*, т.е. функцию, которая принимает один или более параметров (на этапе определения функции их количество неизвестно) и выводит их на отдельных строках.

Начиная с *PHP* 5.6, можно использовать компактный способ организации переменного числа параметров в функции. Для этого перед параметром может быть указано многоточие. Внутри функции такой параметр рассматривается как массив, содержащий все дополнительные параметры.

Пример 2.10. Переменное число параметров.

function myecho(...$planetsArr)

{

foreach ($planetsArr as $planet) {

echo "$planet <br />\n";

}

}

// Отображаем строки одну под другой

myecho("Земля", "Марс", "Меркурий", "Венера");

Оператор "..." может использоваться не только перед аргументами функций, но и при вызове с массивом. Это позволяет осуществить "развертывание" массива. Рассмотрим пример.

Пример 2.11. Использование "...".

function toomanyargs($fst, $snd, $thd, $fth)

{

echo "Первый параметр: $fst<br />";

echo "Второй параметр: $snd<br />";

echo "Третий параметр: $thd<br />";

echo "Четвертый параметр: $fth<br />";

}

// Отображаем строки одну под другой

$planetsArr = ["Земля", "Марс", "Меркурий", "Венера"];

toomanyargs(...$planetsArr);

## 2.2. Области видимости переменных, глобальные переменные

### 2.2.1. Локальные переменные

В примерах из параграфа 2.1 рассматривались аргументы функции как некие временные объекты, которые создаются в момент вызова и исчезают после окончания работы функции. Такими же свойствами будут обладать не только аргументы, но и все другие переменные, инициализируемые или используемые внутри функции. Совокупность таких переменных называют областью видимости внутри функции.

Пример 2.12. Локальные переменные.

function silly()

{

$i = mt\_rand();

// Эта $i не имеет к глобальной $i никакого отношения!

echo "$i<br />";

}

// Выводит в цикле 10 случайных чисел

for ($i = 0; $i != 10; $i++)

silly();

Здесь переменная $*i* в функции будет не той переменной $*i*, которая используется в программе для организации цикла. Поэтому, собственно, цикл и проработает только 10 раз, напечатав 10 случайных чисел.

### 2.2.2. Глобальные переменные

В *PHP* есть способ, посредством которого функции могут обратиться к любой глобальной переменной в программе. Для этого до первого использования внешней переменной нужно объявить ее "глобальной" при помощи инструкции *global*:

Пример 2.13. Глобальные переменные в функции.

$monthesArr = [

1 => "Январь",

2 => "Февраль",

// ...

12 => "Декабрь"

];

/\*\*

\* Возвращает название месяца по его номеру

\* **@global** array $monthsArr - массив с месяцами

\* **@param** int $\_\_number - номер месяца

\* **@return** string - Название месяца

\*/

function getMonthName($\_\_number)

{

global $monthesArr;

return $monthesArr[$\_\_number];

}

// Выводит "Февраль"

echo getMonthName(2);

Массив *$monthesArr*, содержащий названия месяцев, довольно объемен. Поэтому описывать его прямо в функции было бы неудобно – он бы тогда создавался при каждом вызове функции.

### 2.2.3. Инструкция global и массив $GLOBALS

Конструкция *global* $*a* говорит о том, что переменная $*a* является глобальной, т. е. является синонимом глобальной $*a*. Синоним в терминах *PHP* – это ссылка.

Пример 2.14. Использование *global.*

function test()

{

global $a;

$a = 10;

}

Приведенное определение функции *test*() полностью эквивалентно:

Пример 2.15. Использование массива $*GLOBALS.*

function test()

{

$a = &$GLOBALS['a'];

$a = 10;

}

Из второго фрагмента следует, что оператор *unset*($*a*) в теле функции не уничтожит глобальную переменную $*a*, а лишь "отвяжет" от нее ссылку $*a*. Точно то же самое происходит и в первом случае.

Так же обратите внимание, что:

1. Массив $*GLOBALS* изначально является глобальным для любой функции, а также для самой программы. Допустимо его использование не только в теле функции, но и в любом другом месте.
2. С массивом $GLOBALS допустимы не все операции, разрешенные с обычными массивами. А именно, мы не можем: присвоить этот массив какой-либо переменной целиком, используя оператор "=";
3. Передать его функции "по значению" – можно передавать только по ссылке.
4. Добавление нового элемента в $*GLOBALS* равнозначно созданию новой глобальной переменной, а выполнение операции *unset*() для него равносильно уничтожению соответствующей переменной.

### 2.2.4. Статические переменные

Существует еще один вид переменных – статические. Работают они точно так же, как и в *C*, значение статической переменной запоминается.

Пример 2.16. Статические переменные.

function selfCount()

{

static $count = 0;

$count++;

echo $count;

}

for ($i = 0; $i < 5; $i++)

selfCount();

После запуска будет выведена строка 12345. Если теперь убрать слово *static*, то результат изменится на: 11111. Так как переменная $*count* стала локальной, и при каждом вызове функции ее значение не определено (что воспринимается оператором ++ как 0). Ключевое слово *static* сообщает компилятору, что уничтожать указанную переменную для нашей функции между вызовами не надо. В то же время присваивание $*count* = 0 сработает только один раз, а именно при самом первом обращении к функции.

## 2.3. Передача параметров по ссылке, по значению, предопределённые параметры

### 2.3.1. Передача параметров по значению

При передаче параметров функции по значению функции не могут изменить параметр-значение, то есть он доступен функции "только для чтения".

По умолчанию аргументы в функцию передаются по значению.

Пример 2.17. Передача параметров по значению.

function addSomeText($\_\_string)

{

$\_\_string .= 'и кое-что еще.';

}

$str = 'Это строка, ';

addSomeText ($str);

// Выведет 'Это строка, '

echo $str;

### 2.3.1. Передача параметров по ссылке

По умолчанию аргументы в функцию передаются по значению. Если необходимо разрешить функции модифицировать свои аргументы, нужно передавать их по ссылке.

Пример 2.18. Передача аргументов по ссылке.

function addSomeText (&$\_\_string) {

$\_\_string .= 'и кое-что еще.';

}

$str = 'Это строка, ';

addSomeText ($str);

// Выведет 'Это строка, и кое-что еще.'

echo $str;

### 2.3.3. Параметры по умолчанию

Параметры по умолчанию будут подставляться, если не были переданы данные для конкретного аргумента.

Пример 2.19. Задание параметров по умолчанию.

function hello($\_\_text = "Привет")

{

echo $\_\_text;

}

hello('Добрый день');

echo "<br/>";

hello();

Первый вызов функции *hello*('Добрый день') выведет: "Добрый день", второй вызов функции *hello*() без передачи значения так же выведет: "Привет".

Значение по умолчанию для некоторого аргумента указывается справа от его имени через знак равенства. Значения аргументов по умолчанию должны определяться справа налево, причем недопустимо, чтобы после любого из таких аргументов шел обычный аргумент.

## 2.4. Рекурсия

### 2.4.1. Прямая и косвенная рекурсии

В *PHP* поддерживаются рекурсивные вызовы функций, т. е. вызовы функции самой себя. Это удобно для таких задач, как обход всего дерева каталогов вашего сервера (с целью подсчитать суммарный объем, который занимают все файлы).

Программы, которые используют рекурсивные процедуры, отличаются простотой и компактностью текста. Но при этом уступают не рекурсивным по быстродействию и потребляемой памяти. Максимальное число рекурсивных вызовов процедуры без возвратов, которое происходит во время выполнения программы называется глубиной рекурсии.

При каждом вызове любой подпрограммы происходит помещение в системный стек данных этой программы, а именно: точки возврата, значений формальных параметров и локальных констант и переменных, то глубина рекурсии не может быть бесконечной, поскольку в конце концов этот системный стек переполнится.

### 2.4.2. Преимущества и недостатки рекурсии

Рекурсия позволяет создавать код с неизменными переменными, что:

* делает код более читабельным;
* защищает от ошибок типа "действия выполнены в не верном порядке", "использована неинициализированная переменная" и других аналогичных;
* облегчает организацию контроля корректности входных данных;
* позволяет читать код с любого места, не просматривая его весь, отслеживая все изменения переменной;
* облегчает отладку;

В некоторых случаях, рекурсия позволяет более ясно сформулировать идею алгоритма. Однако использовать рекурсию следует разумно. Нужно учитывать то, что часто она ведёт к большим расходам стека и снижению производительности.

### 2.4.3. Замена рекурсии итерацией

Рассмотрим пример рекурсивной функции для нахождения факториала.

Пример 2.20. Пример нахождения факториала рекурсивно.

function factorial($\_\_number)

{

if ($\_\_number <= 0)

return 1;

else

return $\_\_number \* factorial($\_\_number - 1);

}

А теперь перепишем этот пример без использования рекурсии.

Пример 2.21. Пример нахождения факториала.

function factorial($\_\_number)

{

$result = 1;

if ($\_\_number <= 0) {

return $result;

}

for($index = 1; $index <= $\_\_number; $index++) {

$result \*= $index;

}

return $result;

}

Отметим, что любую рекурсивную функцию можно заменить нерекурсивной.

# Глава 3. Массивы

## 3.1. Ассоциативный массив и список

### 3.1.1. Отличия списка и ассоциативного массива

Самым простым массивом является список– массив, ключи которого нумеруются с нуля и идут без пропусков.

Пример 3.1. Способы объявления списка.

$array = array("HTML", "CSS", "PHP");

$array = array(

0 => "HTML",

1 => "CSS",

2 => "PHP",

);

$array[0] = "HTML";

$array[1] = "CSS";

$array[2] = "PHP";

Ассоциативный массив – массив, у которого каждый элемент состоит из пары ключ *=>* значение. Ключ должен быть простого типа (чаще всего является строкой, используемой вместо числового индекса) для доступа к значению элемента, а элементы не обязательно хранятся в строгом порядке линейных индексов. Ассоциативные массивы особенно удобны в ситуациях, когда элементы массива удобнее связывать со словами, а не с числами.

Пример 3.2. Ассоциативный массив.

$array = array(

"foo" => "bar",

"bar" => "foo",

);

//короткий синтаксис массива

$array = [

"foo" => "bar",

"bar" => "foo",

];

### 3.1.2. Типы ключей и значений в массиве

На самом деле, все массивы также являются ассоциативными, в том числе и списки. В *PHP* массивы могут содержать ключи типов integer и string одновременно, т.е. *PHP* не делает различия между индексированными и ассоциативными массивами.

Пример 3.3. Ассоциативный массив со смешанными ключами.

$array = array(

"foo" => "bar",

"bar" => $array,

100 => -100,

-100 => 100,

);

В строковых ключах буквы нижнего и верхнего регистров считаются различными. Ключом может быть абсолютно любая строка, содержащая пробелы, символы перевода строки, нулевые символы. Значение элемента массива может быть любого типа, поддерживаемого PHP, в том числе другим массивом.

### 3.1.3. Многомерные массивы

Так как значением массива может быть другой массив, можно создавать массивы любой размерности.

Пример 3.4. Создание многомерного массива.

$names = [

"Chong" => ["name" => "Tom", "born" => "1937"],

"Mark" => ["name" => "Hugo", "born" => "1963"],

];

Для обращения к элементам многомерного массива можно использовать последовательность операторов "[]":

echo $names["Mark"]["born"]; //будет выведено 1963

### 3.1.4. Вывод содержимого массива

Функция print\_r() выводит удобочитаемую информацию о переменной.

mixed print\_r(mixed $expression [, bool $return = FALSE ]);

*$expression* - выражение для вывода на экран. Если в функцию передаются значения типов string, integer или float, будет напечатано само значение. Если передается массив array, значения будут напечатаны в формате, показывающем ключи и элементы массива.

Пример 3.5. Просмотр структуры через *print\_r*()*.*

$a = array (

'a' => 'apple',

'b' => 'banana',

'c' => array ('x', 'y', 'z')

);

//Вывод структуры

print\_r ($a);

Результат выполнения данного примера:

Array

(

[a] => apple

[b] => banana

[c] => Array

(

[0] => x

[1] => y

[2] => z

)

)

Функция var\_dump() выводит информацию о переменной. Функция отображает структурированную информацию об одном или нескольких выражениях, включая их тип и значение.

void var\_dump(mixed $expression [, mixed $... ]);

*$expression* - переменная, значение которой необходимо отобразить.

Пример 3.6. Просмотр структуры через *var\_dump*().

$a = array(1, 2, array("a", "b", "c"));

//Вывод структуры

var\_dump($a);

Результат выполнения данного примера:

array(3) {

[0]=> int(1)

[1]=> int(2)

[2]=> array(3) {

[0]=> string(1) "a"

[1]=> string(1) "b"

[2]=> string(1) "c"

}

}

Функция var\_export() выводит или возвращает интерпретируемое строковое представление переменной. Функция аналогична var\_dump(), но возвращаемое представление является полноценным *PHP*-кодом.

*$expression* - переменная, которую необходимо экспортировать.

mixed var\_export(mixed $expression [, bool $return = FALSE ]);

Пример 3.7. Просмотр структуры через *var\_export*()*.*

$a = array(1, 2, array ("a", "b", "c"));

//Вывод структуры

var\_export($a);

Результат выполнения данного примера:

array (

0 => 1,

1 => 2,

2 =>

array (

0 => 'a',

1 => 'b',

2 => 'c',

),

)

## 3.2. Операции над массивами

### 3.2.1. Доступ по ключу

Как было показано выше для доступа по ключу используется оператор "[]".

Пример 3.8. Обращение по ключу.

echo $names[42];

echo $names["Mark"];

echo $names["Mark"]["name"];

echo getArray()[1];

### 3.2.2. Размер массива

Определить количество элементов в массиве можно при помощи функции *count*().

Описание:

int count(mixed $array\_or\_countable [, int $mode = COUNT\_NORMAL])

*$array\_of\_countable* – массив или объект.

*$mode* – используется при рекурсивном подсчете элементов массива.

Пример 3.9. Размер массива.

//подсчитаем количество элементов массива

$elementsCount = count($namesList);

### 3.2.3. Добавление элементов в массив

Добавление элементов может производиться в определенное место по индексу.

Пример 3.10. Добавление элемента по индексу.

$names[7] = "Mark";

Аналогично происходит изменение элемента по индексу. Если элемента не существовала в данном индексе, он сохранится, иначе он изменится на новый. Добавление может производиться в конец массива.

Пример 3.11. Добавление элемента в конец массива.

$names[] = "Mark";

### 3.2.4. Проверка существования элементов массива

Функция in\_array() проверяет, присутствует ли в массиве значение.

Описание:

bool in\_array(mixed $needle , array $haystack [, bool $strict = FALSE ]);

Функция ищет в haystack значение needle. Если strict не установлен, то при поиске будет использовано нестрогое сравнение. Если третий параметр strict установлен в TRUE тогда функция ***in\_array*()** также проверит соответствие типов параметра needle и соответствующего значения массива haystack. Возвращает TRUE, если needle был найден в массиве, и FALSE в обратном случае.

Пример 3.12. Проверка элемента.

$os = array("Mac", "Linux", "NT", "Irix");

if (in\_array("Irix", $os)) {

echo "Нашел Irix";

}

else {

echo "Не нашел Irix";

}

Функция isset() определяет, была ли установлена переменная значением, отличным от NULL.

bool isset(mixed $var [, mixed $... ]);

Если переменная была удалена с помощью unset(), то она больше не считается установленной**. *isset*()** вернет FALSE, если проверяемая переменная имеет значение NULL.

Пример 3.13. Проверка элемента.

$var = 'переменная';

if (isset($var)) {

echo "Эта переменная определена.";

}

### 3.2.5. Поиск в масссиве

Функция *array\_search*() осуществляет поиск данного значения в массиве и возвращает ключ первого найденного элемента в случае удачи, в противном случае возвращает FALSE.

Описание:

mixed array\_search(mixed $needle , array $haystack [, bool $strict = FALSE ]);

Если третий параметр strict установлен в TRUE, то функция *array\_search()* будет осуществлять поиск проверяя тип needle в haystack.

Пример 3.14. Поиск элементов.

$array = array('blue', 'red', 'green', 'red');

// $key = 2;

$key = array\_search('green', $array);

// $key = 1;

$key = array\_search('red', $array);

### 3.2.6. Перебор массива с помощью ключей

Сначала вычисляется очередной ключ, затем по нему находится значение элемента массива.

Пример 3.15. Перебор списка.

// Распечатаем имена в столбик

for ($index = 0; $index < count($namesList); $index++)

echo $namesList[$index]."\n";

Отметим, что использование вызовов функций внутри конструкции for, может быть неэффективным. В данном случае функция *count*() будет вызвана столько раз, сколько элементов в массиве, что может привести к снижению производительности.

Рассмотрим, как осуществить перебор ассоциативного массива.

Пример 3.16. Перебор ассоциативного массива.

$birth = [

"Thomas Anderson" => "1962-03-11",

"Keanu Reeves" => "1962-09-02",

];

for (reset($birth); ($k = key($birth)); next($birth))

echo "$k родился {$birth[$k]}<br />";

В данном примере:

* функция reset() устанавливает указатель текущего элемента на первую позицию в массиве;
* функция key() возвращает ключ, который имеет текущий элемент;
* функция next() перемещает текущий элемент на одну позицию вперед.

Таким образом невозможно перебирать массив в двух вложенных циклах или функциях. Второй вложенный цикл for "испортит" положение текущего элемента у первого цикла for. Недостатком является и прерывание цикла при элементе с нулевым ключом. Выражение (*$k = key*($*birth*)) будет равно 0, цикл прервется.

### 3.2.7. Перебор массива с помощью foreach()

Конструкция *foreach*() на каждом шаге цикла может одновремено получить ключ и значение текущего элемента.

Рассмотрим пример перебора значений, хранящихся в массиве.

Пример 3.17. Перебор значений массива с помощью foreach().

$names = [

"Thomas Anderson",

"Keanu Reeves",

];

foreach ($names as $name)

echo "$name<br/>";

Пример 3.18. Полный синтаксис foreach().

$names = [

"Thomas Anderson" => "1962-03-11",

"Keanu Reeves" => "1962-09-02",

];

foreach ($names as $key => $value)

echo "$key родился $value<br/>";

Рекомендуется везде использовать перебор с помощью foreach(), поскольку его использование позволяет предотвратить ошибки, связанные с обращением к несуществующим элементам, например, после удаления элементов в середине массива.

### 3.2.8. Изменение элемента по значению

Во время перебора элементов массива часто возникает задача изменения значений элементов. Рассмотрим пример.

Пример 3.19. Неверное изменение элемента.

//Неправильный способ изменения. $name – это копия.

foreach($names as $name)

if ($name == "Thomas")

$name = "Tom";

В данном примере значения массива $*names* не будут изменены, поскольку в $*name*  на каждом шаге цикла содержится копия.

Существует ссылочный вариант синтаксиса foreach() позволяющий изменять перебираемые элементы.

Вариант, при котором необходимо помнить про оставшиеся ссылки.

Пример 3.20. Изменение через ссылочную копию.

//можно, но опасно

//$name - это ссылка на элемент массива

foreach($names as &$name)

if ($name == "Thomas")

$name = "Tom";

//здесь много кода... очень много... забыли о $name!

//определяем новую переменную $name, это приводит к изменению //последнего элемента в массиве $names, поскольку на него

//указывает ссылка!

$name = "Chong";

После завершения цикла $*name* ссылается на последний элемент массива $*names*, в связи с этим требуется после конца цикла удалить ссылку $*name*. Например, следующим образом:

unset($name);

$name = "Chong";

Для того, чтобы не возникало проблем со ссылками, можно воспользоваться обращением непосредственно к массиву $*names*:

Пример 3.21. Изменение через обращение к массиву.

foreach ($names as $index => $name)

if ($name == "Thomas")

//Изменение происходит в исходном массиве по ключу

$names[$index] = "Tom";

### 3.2.9. Удаление элемента

Чтобы удалить какой-либо элемент массива по ключу или удалить полностью сам массив, используется функция unset().

Описание:

*$var* - удаляемая переменная

void unset ( mixed $var [, mixed $... ] )

Рассмотрим пример.

Пример 3.22. Удаление одного элемента массива.

$fruitsArr = array("Banana", "Apple", "Orange", "Cherry");

//Удаляем банан.

unset($fruitsArr[0]);

//Удаляем весь массив

unset($fruitsArr);

### 3.2.10. Удаление элемента по значению

Рассмотрим типичные неправильные способы удаления элементов массива.

Пример 3.23. Попытка удаления копии элемента.

//Так делать бессмысленно. $fruit – это копия

foreach ($fruitsArr as $fruit)

if ($fruit == "Orange")

unset($fruit);

Пример 3.24. Попытка удаления ссылочной копии.

//Удаление ссылки не приводит к удалению значения в массиве

foreach ($fruitsArr as &$fruit)

if ($fruit == "Orange")

unset($fruit);

Для корректного удаления элементов следует обращаться непосредственно к элементу массива:

Пример 3.25. Удаление элементов массива.

foreach ($fruitsArr as $fruitKey => $fruit)

if ($fruit == "Orange")

//Удаление происходит в исходном массиве

unset($fruitsArr[$fruitKey]);

### 3.2.11. Функции array\_keys(), array\_values()

Функция array\_keys() возвращает все или некоторое подмножество ключей массива в виде списка.

Описание:

array array\_keys (array $array [, mixed $search\_value = NULL [, bool $strict = FALSE ]]);

Функция array\_keys() возвращает числовые и строковые ключи, содержащиеся в массиве array. Если указан необязательный параметр search\_value, функция возвращает только ключи, совпадающие с этим параметром. В обратном случае, функция возвращает все ключи массива array.

Пример 3.26. Использование *array\_keys*().

$array = array(0 => 100, "color" => "red");

print\_r(array\_keys($array));

//Результат выполнения:

Array

(

[0] => 0

[1] => color

)

Пример 3.27. Использование *array\_keys*() с параметром *search\_value*.

$array = array("blue", "red", "green", "blue", "blue");

print\_r(array\_keys($array, "blue"));

//Результат выполнения:

Array

(

[0] => 0

[1] => 3

[2] => 4

)

Функция *array\_values*() выбирает все значения массива в новый список.

Описание:

array array\_values( array $array);

***array\_values*()** возвращает массив со всеми элементами массива array. Она также заново индексирует возвращаемый массив числовыми индексами.

Пример 3.28. Использование *array\_values*().

$array = array("size" => "XL", "color" => "gold");

print\_r(array\_values($array));

//Результат выполнения:

Array

(

[0] => XL

[1] => gold

)

### 3.2.12. Функции *explode*()/*implode*()

Данные функции используются для разбиения строки в массив и склейки массива в строку.

Функция *explode*() получает строку, заданную в ее аргументе $string, и пытается найти в ней подстроки, равные первому аргументу $delimiter. Затем по месту вхождения этих подстрок строка "разрезается" на части, помещаемые в массив-список, который и возвращается.

array explode(string $delimiter, string $string [, int $limit = PHP\_INT\_MAX ]);

Функция *implode*() берет значения массива $list, заданного во втором параметре, и "склеивает" его значения в строку при помощи строки $glue из первого параметра.

string implode(string $glue, array $pieces);

Рассмотрим примеры.

Пример 3.29. Разбиения строки функцией *explode*().

$pizza = "кусок1,кусок2,кусок3,кусок4";

$pieces = explode(",", $pizza);

// кусок2

echo $pieces[1];

Пример 3.30. Объединение функцией *implode*().

$pieces = array("кусок1", "кусок2", "кусок3", "кусок4");

//"кусок1 кусок2 кусок3 кусок4"

$pizza = implode(" ", $pieces);

### 3.2.13. Сериализация и десериализация

Любой тип данных в *PHP* можно упаковать в строку, и не важно, что это: массив или даже объект. Делается это с помощью функции serialize(), которая генерирует пригодное для хранения представление переменной. Это полезно для хранения или передачи данных без потери их типа и структуры.

Описание:

string serialize(mixed $value);

*$value* - значение, которое необходимо сериализовать.

Пример 3.31. Сериализация массива.

$names = [

"Thomas Anderson" => "1962-03-11",

"Keanu Reeves" => "1962-09-02",

];

//Сохранение сериализованного массива в файл

file\_put\_contents("names.txt", serialize($names));

Для превращения сериализованной строки обратно в *PHP*-значение, используйте функцию unserialize( ).

Описание:

mixed unserialize(string $str [, array $options ]);

*$str* - cериализованная строка.

Пример 3.32. Десериализация массива.

//Восстановление массива из прошлого примера

$names = unserialize(file\_get\_contents("names.txt"));

## 3.3. Изменение порядка элементов в массиве

### 3.3.1. Сортировка массива

Функция sort() сортирует список в порядке возрастания, а rsort() – в порядке убывания. Сортировка идет по значениям.

Описание:

*$array* – входной массив.

bool sort(array &$array [, int $sort\_flags = SORT\_REGULAR ]);

bool rsort(array &$array [, int $sort\_flags = SORT\_REGULAR ]);

Пример 3.33. Сортировка списков *sort*().

$a = ["One", "Two", "Three", "Four"];

sort($a);

// Array([0]=>Four [1]=>One [2]=>Three [3]=>Two)

print\_r($a);

А теперь передадим функции sort() не список, а обычный ассоциативный массив.

Пример 3.34. Сортировка ассоциативного массива *sort*().

$a = [

"a" => "Zero",

"b" => "Weapon",

"c" => "Alpha",

"d" => "Processor"

];

sort($a);

//Array([0]=>Alpha [1]=>Processor [2]=>Weapon [3]=>Zero)

print\_r($a);

Как видно из примера, при сортировке с помощью *sort*() были утеряны ключи. Для сохранения ключей можно использовать функцию *asort*(). Функция arsort()выполняет то же самое, однако упорядочивает массив не по возрастанию, а по убыванию.

Описание:

bool asort(array &$array [, int $sort\_flags = SORT\_REGULAR ]);

bool arsort(array &$array [, int $sort\_flags = SORT\_REGULAR ]);

*$array* – входной массив.

Пример 3.35. Сортировка массива по значениям *asort*().

$a = [

"a" => "Zero",

"b" => "Weapon",

"c" => "Alpha",

"d" => "Processor"

];

asort($a);

// Array([c]=>Alpha [d]=>Processor [b]=>Weapon [a]=>Zero)

print\_r($a);

Функция ksort() практически идентична функции asort(), с тем различием, что сортировка осуществляется не по значениям, а по ключам (в порядке возрастания). Сортирует массив по ключам, сохраняя отношения между ключами и значениями. Функция для сортировки по ключам в обратном порядке называется *krsort*() и применяется точно в таком же контексте, что и *ksort*()*.*

Описание:

bool ksort(array &$array [, int $sort\_flags = SORT\_REGULAR ]);

bool krsort(array &$array [, int $sort\_flags = SORT\_REGULAR ]);

*$array* – входной массив.

Пример 3.36. Сортировка массива по ключам *ksort*().

$a = [

"c" => "Alpha",

"a" => "Zero",

"d" => "Processor",

"b" => "Weapon",

];

ksort($a);

// Array([a]=>Zero [b]=>Weapon [c]=>Alpha [d]=>Processor)

print\_r($a);

### 3.3.2. "Естественная" сортировка

Функция natsort()реализует алгоритм сортировки, при котором порядок буквенно-цифровых строк будет привычным для человека. Такой алгоритм называется "естественная" сортировка. Функция natcasesort() работает точно так же, как natsort(), однако при сортировке она не учитывает регистр букв.

Описание:

bool natsort(array &$array);

bool natcasesort(array &$array);

*$array* – входной массив.

Пример 3.37. Естественная сортировка *natsort*().

$files = [

"img10.gif",

"img1.gif",

"img2.gif",

"img20.gif",

];

natsort($files);

echo "<pre>"; print\_r($files);

//Результат выполнения:

Array (

[1] => img1.gif

[2] => img2.gif

[0] => img10.gif

[3] => img20.gif

)

### **3.3.3. Пользовательская сортировка: usort, uasort, uksort**

Функция usort() сортирует элементы массива, используя для сравнения значений *callback*-функцию, предоставленную пользователем. Используйте эту функцию, если необходимо отсортировать массив по какому-нибудь необычному признаку. usort() не сохраняет связей между ключами и значениямии потому пригодна только для сортировки списков.

Описание:

bool usort(array &$array , callable $value\_compare\_func);

*$array* – входной массив.

*$value\_compare\_func* – функция сравнения.

Пример 3.38. Использование *usort*() с многомерным массивом.

$fruits = [

['fruit' => 'lemon', 'color' => 'yellow'],

['fruit' => 'apple', 'color' => 'red'],

['fruit' => 'grape', 'color' => 'green']

];

//функция сравнивает два элемента по подключу "fruit"

function cmp($\_\_a, $\_\_b)

{

return strcmp($a["fruit"], $b["fruit"]);

}

usort($fruits, "cmp");

В результате верхний уровень массива $*fruits* будет отсортирован по возрастанию поля "fruit".

Функция uasort() cортирует массив, используя пользовательскую функцию для сравнения элементов с сохранением ключей.

Функция uksort() cортирует массив по ключам, используя пользовательскую функцию для сравнения ключей. Cвязи между ключами и значениями функции uksort() сохраняются.

### 3.3.4. Сортировка многомерного массива: array\_multisort()

Функция array\_multisort() может быть использована для сортировки сразу нескольких массивов или одного многомерного массива. Массивы, которые передаются функции array\_multisort(), должны содержать одинаковое количество аргументов и все вместе воспринимаются как своеобразная таблица. Сортировке подвергается лишь первый массив, а значения последующих массивов выстраиваются в соответствии с ним.

Описание:

bool array\_multisort(array &$ar1 [,$arg [, ... [, ...]]] );

*$ar1* – сортируемый массив.

Пример 3.39. Использование функции *array\_multisort*().

$arr1 = [3, 4, 2, 7, 1, 5];

$arr2 = ["world", "Hello", "yes", "no", "apple", "wet"];

array\_multisort($arr1, $arr2);

print\_r($arr1);

print\_r($arr2);

//Результат выполнения:

//первый массив

Array (

[0] => 1

[1] => 2

[2] => 3

[3] => 4

[4] => 5

[5] => 7

)

//второй массив

Array (

[0] => apple

[1] => yes

[2] => world

[3] => Hello

[4] => wet

[5] => no

)

Первый массив был отсортирован по возрастанию, а элементы второго массива выстроены в соответствии с первым так, как если бы между элементами двух массивов существовала связь.

### 3.3.5. Переворачивание массива

Функция array\_reverse() возвращает массив с элементами, расположенными в обратном порядке.

Описание:

array array\_reverse(array $array [, $preserve\_keys = FALSE ]);

*$array* – входной массив.

Пример 3.40. Использование *array\_reverse*().

$input = array("php", 4.0, array("green", "red"));

$reversed = array\_reverse($input);

print\_r($reversed);

//Результат выполнения:

Array

(

[0] => Array

(

[0] => green

[1] => red

)

[1] => 4

[2] => php

)

### 3.3.6. Перемешивание элементов: shuffle()

Функция shuffle() "перемешивает" список, переданный ей первым параметром, так, чтобы его значения распределялись случайным образом. Обратите внимание, что, во-первых, изменяется сам массив, а во-вторых, ассоциативные массивы воспринимаются как списки.

Описание:

bool shuffle(array &$array);

*$array* – входной массив.

Пример 3.41. Перемешивание списка.

$concept = ["Banana", "Coffee", "Ice cream", "Throat"];

shuffle($concept);

print\_r($concept);

Приведенный фрагмент выводит массив в случайном порядке.

## 3.4. Работа с массивами как с множествами

Списки можно рассматривать, как множества элементов. В *PHP* существуют функции для выполнения основных теоретико-множественных операций с такими списками (пересечение, разность, объединение и др.).

### 3.4.1. Пересечение

Функция array\_intersect() представляет собой операцию "пересечения" множеств. Она возвращает те значения массива $*array1*, которые присутствуют также и в $*array2*, $*array3* и т. д.

Описание:

array array\_intersect(array $array1 , array $array2 [, array $... ]);

Пример 3.42. Пересечение множеств.

$native = ["green", "red", "blue"];

$colors = ["red", "yellow", "green", "cyan", "black"];

$inter = array\_intersect($colors, $native);

// Array([0]=>red [2]=>green)

print\_r($inter);

Связь между ключами и значениями данная функция сохраняет. Функция возвращает не список, а именно ассоциативный массив*.*

### 3.4.2. Разность

Функция array\_diff() осуществляет теоретико-множественную операцию "разность множеств". Она возвращает массив, составленный из значений *$array1*, не входящих ни в один из массивов *$array2*, *$array3* и т. д.

Описание:

array array\_diff(array $array1 , array $array2 [, array $... ]);

Пример 3.43. Разность множеств.

$array1 = array("a" => "green", "red", "blue", "red");

$array2 = array("b" => "green", "yellow", "red");

$result = array\_diff($array1, $array2);

print\_r($result);

//Результат выполнения:

Array

(

[1] => blue

)

Очевидно, что данную функцию можно использовать для сравнения массивов, для это достаточно сравнить разность с пустым массивом.

### 3.4.3. Объединение

К сожалению, в *PHP* нет специальной функции для объединения множеств. Тем не менее ее легко реализовать при помощи array\_merge() и функции array\_unique().

Описание:

array array\_merge(array $array1 [, array $... ]);

array\_merge() cливает элементы одного или большего количества массивов таким образом, что значения одного массива присоединяются к концу предыдущего. Результатом работы функции является новый массив.

Если входные массивы имеют одинаковые строковые ключи, тогда каждое последующее значение будет заменять предыдущее. Однако, если массивы имеют одинаковые числовые ключи, значение, упомянутое последним, не заменит исходное значение, а будет добавлено в конец массива.

В результирующем массиве значения исходного массива с числовыми ключами будут перенумерованы в возрастающем порядке, начиная с нуля.

Пример 3.44. Использование функции *array\_merge*().

$array1 = array("color" => "red", 2, 4);

$array2 = array("a", "b", "color" => "green", "shape" => "trapezoid", 4);

$result = array\_merge($array1, $array2);

print\_r($result);

//Результат выполнения:

Array

(

[color] => green

[0] => 2

[1] => 4

[2] => a

[3] => b

[shape] => trapezoid

[4] => 4

)

Рассмотрим функцию *array\_unique*().

array array\_unique(array $array [, int $sort\_flags = SORT\_STRING ]);

*array\_unique*() убирает повторяющиеся значения из массива. Обратите внимание, что ключи сохранятся.

Пример 3.45. Использование функции *array\_unique*().

$input = array("a" => "green", "red", "b" => "green", "blue", "red");

$result = array\_unique($input);

print\_r($result);

//Результат выполнения:

Array

(

[a] => green

[0] => red

[1] => blue

)

Данная функция может использоваться совместно с array\_merge(), чтобы получить теоретико-множественную операцию "объединение".

Пример 3.46. Объединение множеств.

$native = ["green", "red", "blue"];

$colors = ["red", "yellow", "green", "cyan"];

$inter = array\_unique(array\_merge($colors, $native));

print\_r($inter);

//Результат выполнения:

Array

(

[0] => red

[1] => yellow

[2] => green

[3] => cyan

[6] => blue

)

Рассмотрим так же функцию array\_merge\_recursive(), которая осуществляет рекурсивное слияние двух или более массивов.

Описание:

array array\_merge\_recursive (array $array1 [, array $... ]);

Функция array\_merge\_recursive() сливает элементы двух или более массивов таким образом, что значения одного массива присоединяются в конец другого. Если входные массивы имеют одинаковые строковые ключи, то значения этих ключей сливаются в массив, и это делается рекурсивно, так что если одно из значений является массивом, то функция сливает его с соответствующим значением в другом массиве.

Пример 3.47. Использование *array*\_*merge*\_*recursive*.

$ar1 = array("color" => array("favorite" => "red"), 5);

$ar2 = array(10, "color" => array("favorite" => "green", "blue"));

$result = array\_merge\_recursive($ar1, $ar2);

print\_r($result);

//Результат выполнения:

Array

(

[color] => Array

(

[favorite] => Array

(

[0] => red

[1] => green

)

[0] => blue

)

)

### 3.4.4. Разбиение массива на части

Функция *array\_chunk*() разбивает массив на части.

Описание:

array array\_chunk(array $array , int $size [, bool $preserve\_keys = FALSE ]);

Разбивает массив $array на несколько массивов размером в $size элементов. Последний массив из полученных может содержать меньшее количество значений, чем указано в $size.

Возвращает многомерный массив с числовыми индексами, начинающимися с нуля, каждый элемент которого содержит $size элементов из оригинального массива.

Если $preserve\_keys установлено в TRUE, ключи оригинального массива будут сохранены. По умолчанию установлено в FALSE, что переиндексирует каждую часть с числовыми ключами

Пример 3.48. Разбиение множеств.

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

print\_r(array\_chunk($array, 2, true));

//Результат выполнения:

Array

(

[0] => Array

(

[0] => a

[1] => b

)

[1] => Array

(

[2] => c

[3] => d

)

[2] => Array

(

[4] => e

)

)

## 3.5. Функции для работы с массивами

В таблице 3.1. кратко перечислены основные функции для работы с массивами.

**Таблица 3.1.** Функции для работы с массивами.

|  |  |
| --- | --- |
| Название | Описание |
| *print\_r* | Выводит удобочитаемую информацию о массиве. |
| *var\_dump* | Выводит информацию о массиве. |
| *var\_export* | Выводит строковое представление массива. |
| *array\_change\_key\_case* | Меняет регистр всех ключей в массиве. |
| *array\_chunk* | Разбивает массив на части. |
| *array\_column* | Возвращает массив из значений одного столбца входного массива. |
| *array\_combine* | Создает новый массив, используя один массив в качестве ключей, а другой для его значений. |
| *array\_count\_values* | Подсчитывает количество всех значений массива. |
| *array\_diff\_assoc* | Вычисляет разницу между массивами с дополнительной проверкой индекса. |
| *array\_diff\_key* | Вычисляет разницу между массивами, сравнивая ключи. |
| *array\_diff\_uassoc* | Вычисляет разницу между массивами с дополнительной проверкой индекса, осуществляемой при помощи *callback*-функции. |
| *array\_diff\_ukey* | Вычисляет разницу между массивами, используя *callback*-функцию для сравнения ключей. |
| *array\_diff* | Вычисляет разницу между массивами. |
| *array\_fill\_keys* | Создает массив и заполняет его значениями, с определенными ключами. |
| *array\_fill* | Заполняет массив значениями. |
| *array\_filter* | Фильтрует элементы массива с помощью *callback*-функции. |
| *array\_flip* | Меняет местами ключи с их значениями в массиве. |
| *array\_intersect\_assoc* | Вычисляет пересечение массивов с дополнительной проверкой индекса. |
| *array\_intersect\_key* | Вычисляет пересечение массивов, сравнивая ключи. |
| *array\_intersect\_uassoc* | Вычисляет пересечение массивов с дополнительной проверкой индекса, осуществляемой при помощи *callback*-функции. |
| *array\_intersect\_ukey* | Вычисляет пересечение массивов, используя *callback*-функцию для сравнения ключей. |
| Название | Описание |
| *array\_intersect* | Вычисляет пересечение массивов. |
| *array\_key\_exists* | Проверяет, присутствует ли в массиве указанный ключ или индекс. |
| *array\_keys* | Возвращает все или некоторое подмножество ключей массива. |
| *array\_map* | Применяет *callback*-функцию ко всем элементам указанных массивов. |
| *array\_merge\_recursive* | Рекурсивное слияние двух или более массивов. |
| *array\_merge* | Сливает один или большее количество массивов. |
| *array\_multisort* | Сортирует несколько массивов или многомерные массивы. |
| *array\_pad* | Дополняет массив определенным значением до указанной длины. |
| *array\_pop* | Извлекает последний элемент массива. |
| *array\_product* | Вычисляет произведение значений массива. |
| *array\_push* | Добавляет один или несколько элементов в конец массива. |
| *array\_rand* | Выбирает одно или несколько случайных значений из массива. |
| *array\_reduce* | Итеративно уменьшает массив к единственному значению, используя *callback*-функцию. |
| *array\_replace\_recursive* | Рекурсивно заменяет элементы первого массива элементами переданных массивов. |
| *array\_replace* | Заменяет элементы массива элементами других переданных массивов. |
| *array\_reverse* | Возвращает массив с элементами в обратном порядке. |
| *array\_search* | Осуществляет поиск данного значения в массиве и возвращает ключ первого найденного элемента в случае удачи. |
| *array\_shift* | Извлекает первый элемент массива. |
| *array\_slice* | Делает срез массива. |
| *array\_splice* | Удаляет часть массива и заменяет её чем-нибудь ещё. |
| *array\_sum* | Вычисляет сумму значений массива. |
| *array\_udiff\_assoc* | Вычисляет расхождение в массивах с дополнительной проверкой индексов, используя для сравнения значений *callback*-функцию. |
| *array\_udiff\_uassoc* | Вычисляет разницу между массивами с дополнительной проверкой индексов, используя для сравнения значений и индексов *callback*-функцию. |
| *array\_udiff* | Вычисляет разницу между массивами, используя для сравнения *callback*-функцию. |
| *array\_uintersect\_assoc* | Вычисляет пересечение массивов с проверкой индексов, используя *callback*-функцию. |
| *array\_uintersect\_uassoc* | Вычисляет пересечение массивов с дополнительной проверкой индекса, используя для сравнения индексов и значений индивидуальные *callback*-функции. |
| *array\_uintersect* | Вычисляет пересечение массивов, используя для сравнения значений *callback*-функцию. |
| *array\_unique* | Убирает повторяющиеся значения из массива. |
| *array\_unshift* | Добавляет один или несколько элементов в начало массива. |
| *array\_values* | Выбирает все значения массива. |
| *array\_walk\_recursive* | Рекурсивно применяет пользовательскую функцию к каждому элементу массива. |
| Название | Описание |
| *array\_walk* | Применяет заданную пользователем функцию к каждому элементу массива. |
| *array* | Создает массив. |
| *arsort* | Сортирует массив в обратном порядке, сохраняя ключи. |
| *asort* | Сортирует массив, сохраняя ключи. |
| *compact* | Создает массив, содержащий названия переменных и их значения. |
| *count* | Подсчитывает количество элементов массива или что-то в объекте. |
| *current* | Возвращает текущий элемент массива. |
| *end* | Устанавливает внутренний указатель массива на его последний элемент. |
| *extract* | Импортирует переменные из массива в текущую таблицу символов. |
| *in\_array* | Проверяет, присутствует ли в массиве значение. |
| *key\_exists* | Псевдоним *array*\_*key*\_*exists*(). |
| *key* | Выбирает ключ из массива. |
| *krsort* | Сортирует массив по ключам в обратном порядке. |
| *ksort* | Сортирует массив по ключам. |
| *list* | Присваивает переменным из списка значения подобно массиву |
| *natcasesort* | Сортирует массив, используя алгоритм *"natural order"* без учета регистра символов. |
| *natsort* | Сортирует массив, используя алгоритм *"natural order"*. |
| *next* | Перемещает указатель массива вперед на один элемент. |
| *pos* | Псевдоним *current*(). |
| *prev* | Передвигает внутренний указатель массива на одну позицию назад. |
| *range* | Создает массив, содержащий диапазон элементов. |
| *reset* | Устанавливает внутренний указатель массива на его первый элемент. |
| *rsort* | Сортирует массив в обратном порядке. |
| *shuffle* | Перемешивает массив. |
| *sizeof* | Псевдоним *count*(). |
| *sort* | Сортирует массив. |
| *uasort* | Сортирует массив, используя пользовательскую функцию для сравнения элементов с сохранением ключей. |
| *uksort* | Сортирует массив по ключам, используя пользовательскую функцию для сравнения ключей. |
| *usort* | Сортирует массив по значениям используя пользовательскую функцию для сравнения элементов. |

# Глава 4. Строки

## 4.1. Кодировки

### 4.1.1. ASCII

При создании ранних языков программирования и программного обеспечения долгое время использовалась кодировка *ASCII*, в которой каждый байт соответствовал одному символу. Изначально кодировка содержала 127 символов. Например 10 символ – перевод строки, 65 символ соответствует английской букве *A*. Первые 32 символа относятся к управляющим символам, в то время как далее следуют видимые символы. Коды этих символов до сих пор актуальны, они могут быть распечатаны с помощью функции *chr*(). Для функции *chr*() в *PHP* существует обратная функция *ord*(), которая по *ASCII*-коду возвращает символ.

Кодировка *ASCII* – однобайтовая. В байте хранится 8 бит, каждый бит может принимать лишь два значения: либо 0, либо 1. Таким образом, в байте может храниться 256 символов. То есть оригинальная кодировка *ASCII* занимает лишь половину кодов, которые можно закодировать одним байтом. Дополнительные 127 кодовых позиций использовались для специальных задач. Например, для контроля четности при передаче информации по сети или для кодирования символов других языков.

Стандарты кодирования языков появились поздно, к моменту их возникновения в некоторых языках появилось сразу несколько интенсивно используемых кодировок. До момента массового перехода на кодировку *UTF-8* в русском языке их было 5 штук:

* *KOI8-R;*
* *Windows-1251*;
* *ISO8859-5*;
* *CP866*;
* *MAC-Cyrilic*.

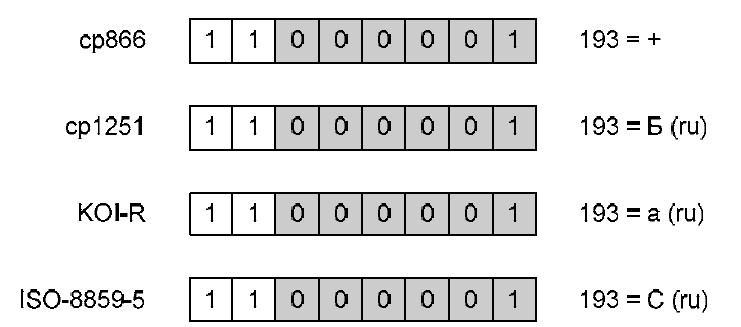


Рис. 4.1 – В различных кодировках один и тот же код обозначает разные символы.

В каждой из таких кодировок коды соответствовали разным символам языка.

Используя однобайтовую кодировку, нельзя было одновременно отобразить русский и немецкий тексты. Кроме того, в 127 символов можно было закодировать лишь какой-то один язык или языковую группу, использующую один не слишком большой алфавит, например кириллицу. Для кодирования иероглифов азиатских языков 127 символов было слишком мало. Для их кодирования приходилось использовать два и более байта.

### 4.1.2. Unicode

Для устранения проблем нехватки символов кодировок используется многобайтовая кодировка *Unicode.*

В *Unicode* большой русской букве А назначается код *U+0410*, маленькой русской букве я – код *U+044F*. Последовательность *U*+ означает *Unicode*, за которым следует код символа в шестнадцатеричном формате. Видно, что под хранение каждой русской буквы требуется 2 байта. *Unicode* сохраняет обратную совместимость с *ASCII*.

Двумя байтами можно закодировать лишь 65 632 символа, для хранения всех симолов этого оказывается недостаточно, поэтому иногда используют Unicode-кодировки размером в 4 байта. Кроме того, например, двухбайтовый *Unicode*, сохраняя обратную совместимость *с ASCII*, вынуждает на каждый символ английского текста сохранять пустой дополнительный байт, поэтому эти кодировки иногда называют "широкими".

### 4.1.3. UTF-8

В этой кодировке каждый код от 0 до 127 записывается в один байт, а все последующие коды в 2, 3 и более байтов. Данная особенность привела к полной обратной совместимости с *ASCII*, т. е. английские и управляющие символы в *ASCII* можно трактовать как *UTF-8*.

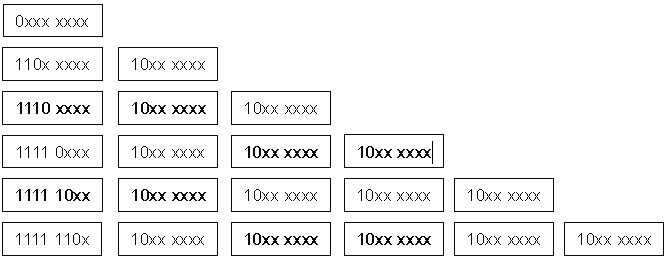


Рис. 4.2 – Организация хранения данных в UTF-8.

На рис. 4.2 представлена организация кодов в *UTF-8*, если код начинается с нулевого бита, он занимает один байт и соответствует символу из *ASCII*. Если первые два бита равны единице, а последующий равен нулю (110), это соответствует символу, который занимает 2 байта, префикс 1110 соответствует трем байтам и т. д. При этом последующие байты начинаются с последовательности бит 10. С помощью такой схемы кодировку можно расширять бесконечно, и она всегда останется совместимой с *ASCII*.

В *PHP* 7 символы из кодировки *UTF-8* могут быть заданы при помощи специального синтаксиса, в строках указывается последовательность *\u*, после которой следует шестнадцатеричный код символа в фигурных скобках.

Пример 4.1. Вывод *UTF-8* русской буквы А.

<?php

echo "\u{0410}";

Все современное программное обеспечение создается в расчете на кодировку *UTF-8*, в которой английские символы занимают один байт, а русские – два. Однако PHP создавался задолго до перехода на *UTF-8*. В результате, при обращении к строке в кодировке *UTF-8*, как к массиву символов, используя квадратные скобки, в случае английского языка получается символ, а в случае русского – только половина символа.

На практике для решения проблемы, как правило, подключают расширение *mbstring*, поддерживающее работу с многобайтными кодировками и либо используют функции *mbstring* напрямую, либо настраивают *PHP* таким образом, чтобы стандартные строковые функции *PHP* заменялись *mbstring*-аналогами.

## 4.2. Операции со строками

### 4.2.1. Конкатенация строк

Для конкатенации строк в *PHP* используется специальный оператор "." (точка). Оператор "." всегда воспринимает свои операнды как строки и возвращает строку. Если один из операндов не может быть переведен в строковое представление, т.е. если это массив или объект, то он воспринимается как строки *array* и *object* соответственно.

### 4.2.2. Сравнение строк

Если используются операторы сравнения "==" и "!=" (или любые другие, которые могут потребовать перевода строки в число) с операндами-строками, то результат не всегда оказывается очевидным.

В операциях сравнения пустая строка "" в первую очередь трактуется как 0 (ноль), во вторую как "пусто". Операнды сравниваются как строки только в том случае, если они оба – строки. Если же хотя бы один из операндов – не строка, но может трактоваться как *false*, *PHP* использует логический контекст при сравнении. Пустая строка воспринимается как *false*, а *false* == 0, поэтому получается приведенный выше результат. Для избежания неоднозначных ситуаций используйте оператор "===", в таком случае сравнение будет производиться и по значению, и по их типу.

Пример 4.2. Особенности операторов сравнения.

<?php

$one = 1;

$zero = 0;

// Очевидно, не равно - не выводит 1

if ($one == "") echo 1;

// Внимание! печатает 2!

if ($zero == "") echo 2;

// Правильный ответ

if ("$zero" == "") echo 3;

// Тоже правильный ответ

if (strval($zero) == "") echo 4;

// Лучше делать так:

if ($zero === "") echo 5;

### 4.2.3. Работа с однобайтовыми строками

Длина строки может быть получена с помощью функции *strlen*().

int strlen(string $st)

Данная функция возвращает просто длину строки в байтах, обратите внимание, что для многобайтовой кодировки использование этой функции не даст возможности получить размер строки в символах, поскольку некоторые символы могут быть больше одного байта.

В *PHP* строки рассматриваются как массивы, элементами которых выступают символы. Точно так же, как в числовых массивах, первый элемент помечается индексом 0, второй 1 и т.д.

Пример 4.3. Обращение к строке, как к символьному массиву.

<?php

$str = "PHP";

echo $str[0]."<br>"; //P

echo $str[1]."<br>"; //H

echo $str[2]."<br>"; //P

Если указать в квадратных скобках два значения, разделенных запятой, из строки можно извлекать подстроки. Очевидно, что обращение по индексу имеет смысл только для однобайтовой строки.

Пример 4.4. Извлечение подстрок из строки.

<?php

$str = "Hello, world!";

echo $str[0, 4]; //Hello

#### Сравнение строк

Сравнение строк можно осуществлять при помощи оператора "==". При сравнении учитывается регистр строк, поэтому для того, чтобы сравнение строк осуществлялось без учета регистра, необходимо выполнить приведение строк к верхнему регистру при помощи *strtoupper*() или к нижнему с использованием функции *strtolower*().

Также можно использовать для сравнения функцию:

int strcmp(string $str1, string $str2)

Функция сравнивает строки без учета регистра. Возвращает 0, если строки равны, отрицательное число, если $*str1* меньше $*str2*, и положительное, если $*str1* больше $*str2*. Сравниваться могут, в том числе и бинарные данные.

Пример 4.5. Сравнение строк.

<?php

$str = "Cтрока текста";

if ($str == "строка текста") {

echo "строки равны<br>";

}

else {

echo "Строки не равны<br>";

}

if (strtoupper($str) == "строка текста") {

echo "строки равны<br>";

}

else {

echo "Строки не равны<br>";

}

#### Поиск в строке

Поиск в строке осуществляется с помощью функции *strpos*().

int strpos(string $where, string $what [, int $from = 0])

Функция ищет в строке $*where* подстроку $*what* и в случае успеха возвращает позицию (индекс) этой подстроки в строке. Первый символ строки имеет индекс 0. Необязательный параметр $*from* можно задавать, если поиск нужно вести не с начала строки $*where*, а с какой-то другой позиции. В этом случае следует позицию передать в $*from*. Если подстроку найти не удалось, функция возвращает *false*. Для проверки результата вызова *strpos*() на *false* – рекомендуется использовать только оператор "===". Типовая ошибка при использовании strpos() показана в примере 4.6.

Пример 4.6. Неправильное использование *strpos*().

<?php

$str = "Cтрока текста";

// не найдет

if (strpos($str, "C")) {

echo "есть совпадение";

}

// этот if полностью эквивалентен прошлому

if (strpos($str, "C") != false) {

echo "есть совпадение";

}

// а вот так правильно

if (strpos($str, "C") !== false) {

echo "есть совпадение";

}

Функция *stripos*() аналогична функции *strpos*() за исключением того факта, что в процессе поиска не учитывается регистр.

Функция *strrpos*() производит поиск с конца строки.

Функция *strpbrk*() ищет в строке $*str* любой символ из заданного набора $*char\_list* и возвращает подстроку, начиная с первого найденного символа и до конца строки. Возвращает *false*, если ни один символ не был найден.

string strpbrk (string $str, string $char\_list)

Пример 4.7. Использование *strpbrk*().

<?php

echo strpbrk("Hello world", 'wl'); //llo world

#### Подрезка строки

Для подрезки строки используется функция *substr*().

string substr(string $str, int $start [,int $length])

Назначение этой функции – возвращать участок строки $*str,* начиная с позиции $*start* и длиной $*length*. Если $*length* не задана, то подразумевается подстрока от $*start* до конца строки $*str*. Если передать в $*start* отрицательное число, то будет считаться, что это число является индексом подстроки, но только отсчитываемым от конца $*str* (например, −1 означает "начиная с последнего символа строки"). Параметр $*length*, если он задан, тоже может быть отрицательным. В этом случае последним символом возвращенной подстроки будет символ из $*str* с индексом $*length*, определяемым от конца строки.

Функция *strtok*() возвращает часть строки $*str* до разделителя $*token*. При последующих вызовах функции возвращается следующая часть до следующего разделителя, и так до конца строки.

string strtok(string $str, string $token)

При первом вызове функция принимает два аргумента: исходную строку $*str* и разделитель $*token*, при каждом последующем вызове указывать не следует, иначе будет возвращаться первая часть строки.

#### Замена подстроки

Функия *str\_replace*() заменяет в строке $*text* все вхождения подстроки $*from* (с учетом регистра) на $*to* и возвращает результат.

string str\_replace(string $from, string $to, mixed $text [, int &$count])

Исходная строка, переданная третьим параметром, при этом не меняется. Если указан необязательный параметр $*count*, в него будет записано количество произведенных замен. Эта функция работает значительно быстрее, чем более универсальная *preg\_replace*() и ее часто используют, если нет необходимости в сложных правилах поиска подстроки.

Параметр $*text* описан выше как *mixed,* чтобы было возможно передавать целый массив строк, а не только одну – единственную строку. Если $*text* – массив, то замена производится в каждом его элементе, а возвращает функция результирующий список.

Функция *str\_ireplace*() работает так же, как *str\_replace*(), но только заменяет строки без учета регистра символов.

Функция *substr\_replace*() предназначена для замены в строке $*text* участка, начинающегося с позиции $*start* и длины $*len*. Этот участок заменяется значением параметра $*to*.

string substr\_replace(string $text, string $to, int $start [, int $len])

#### Подстановка

Функции подстановки предназначены для того, чтобы в одном и том же тексте искать и заменять сразу несколько пар различных подстрок.

У рассмотренной ранее в этом параграфе функции *str\_replace*() первые два параметра должны иметь строковый тип. Однако функция *str\_replace*() значительно мощнее и позволяет передавать в качестве аргументов целые массивы строк. Если $*from* и $*to* – массивы, замена происходит так: каждый элемент из $*from* заменяется соответствующим (по номеру) элементом из $*to*. Таким образом можно за один вызов функции заменять сразу несколько пар подстрок.

Так же для подстановок может быть использована функция *strtr*().

string strtr(string $str, string $from, string $to)

Эта функция в строке $*str* заменяет все символы, встречающиеся в $*from*, их "парными" (т. е. расположенными в тех же позициях, что и во $*from*) из $*to*.

У функции *strtr*() существует две разновидности: первая – с тремя параметрами (рассмотрена выше), а вторая – с двумя.

string strtr(string $str, array $substitutes)

Функция *strtr*() с двумя параметрами берет строку $*str* и проводит в ней контекстный поиск и замену: ищутся подстроки – ключи в массиве $*substitutes* – и замещаются на соответствующие им значения. Функцию *strtr*() удобно использовать для перевода русского текста в так называемый "транслит".

Пример 4.8. Транслитерация строк.

<?php

function transliterate($st)

{

//первый вариант strtr()

$st = strtr($st,

"абвгдежзийклмнопрстуфыэАБВГДЕЖЗИЙКЛМНОПРСТУФЫЭ",

"abvgdegziyklmnoprstufyeABVGDEGZIYKLMNOPRSTUFYE"

);

//второй вариант strtr()

$st = strtr($st, array(

'ё'=>"yo", 'х'=>"h", 'ц'=>"ts", 'ч'=>"ch", 'ш'=>"sh",

'щ'=>"shch", 'ъ'=>'', 'ь'=>'', 'ю'=>"yu", 'я'=>"ya",

'Ё'=>"Yo", 'Х'=>"H", 'Ц'=>"Ts", 'Ч'=>"Ch", 'Ш'=>"Sh",

'Щ'=>"Shch", 'Ъ'=>'', 'Ь'=>'', 'Ю'=>"Yu", 'Я'=>"Ya"

));

return $st;

}

echo transliterate("У попа была собака, он ее любил.");

Функция *strtr*() несовместима с *UTF-8* и работает только с однобайтовыми кодировками. В расширении *mbstring* аналог для нее не предусмотрен. Поэтому при работе с кодировкой *UTF-8* используется вариант с функцией *str\_replace*().

Пример 4.9. Транслитерация строк c *str\_replace*().

<?php

function transliterate($st)

{

$pattern = ['а', 'б', 'в', 'г', 'д', 'е', 'е',

'ж', 'з', 'и', 'й', 'к', 'л', 'м',

'н', 'о', 'п', 'р', 'с', 'т', 'у',

'ф', 'х', 'ч', 'ц', 'ш', 'щ', 'ъ',

'ы', 'ь', 'э', 'ю', 'я',

'А', 'Б', 'В', 'Г', 'Д', 'Е', 'Е',

'Ж', 'З', 'И', 'Й', 'К', 'Л', 'М',

'Н', 'О', 'П', 'Р', 'С', 'Т', 'У',

'Ф', 'Х', 'Ч', 'Ц', 'Ш', 'Щ', 'Ъ',

'Ы', 'Ь', 'Э', 'Ю', 'Я'];

$replace = ['a', 'b', 'v', 'g', 'd', 'e', 'yo',

'zh', 'z', 'i', 'y', 'k', 'l', 'm',

'n', 'o', 'p', 'r', 's', 't', 'u',

'f', 'h', 'ch', 'ts', 'sh', 'shch', '\'',

'y', '', 'e', 'yu', 'ya',

'A', 'B', 'V', 'G', 'D', 'E', 'Yo',

'Zh', 'Z', 'I', 'Y', 'K', 'L', 'M',

'N', 'O', 'P', 'R', 'S', 'T', 'U',

'F', 'H', 'CH', 'Ts', 'Sh', 'Shch', '\'',

'Y', '', 'E', 'Yu', 'Ya'];

return str\_replace($pattern, $replace, $st);

}

echo transliterate("У попа была собака, он ее любил.");

Функция *strtr*() начинает поиск с самой длинной подстроки и не проходит по одному и тому же ключу дважды. Это отличает ее от *str\_replace*().

Пример 4.10. Различия между *strtr*() и *str\_replace*().

<?php

$text = "matrix has you";

$repl = ["matrix" => "you", "you" => "matrix"];

echo "str\_replace(): ".

str\_replace(array\_keys($repl), array\_values($repl), $text)."\n";

echo "strtr(): ".

strtr($text, $repl);

//Результаты выполнения программы:

str\_replace(): matrix has matrix

strtr(): you has matrix

Функция *str\_replace*() в цикле перебирает свой первый аргумент-массив и заменяет найденные подстроки:

1. На первой итерации заменяется "*matrix*" => "*you*". Текст принимает вид: "*you has you*".
2. На второй итерации заменяется "*you*" => "*matrix*". Текст принимает вид: "*matrix has matrix*". Функция не знает, что первое "*you*" заменять не надо, потому что оно было получено на предыдущей итерации.

Функция *strtr*() всегда заменяет самые длинные подстроки и никогда не делает замен в тексте, полученном в результате предыдущей подстановки. За то, что *strtr*() работает именно так, приходится расплачиваться: если ключи массива замен длинные (а точнее, сильно различаются по длине между собой), замена происходит очень медленно – гораздо медленнее, чем при использовании *str\_replace*().

Рекомендуется использовать функцию *str\_replace*() в случае, если заменяемые значения не будут перекрываться с результатом предыдущих замен. Использовать функцию *strtr*() следует во всех остальных случаях. Если работа ведется с русским текстом в *UTF-8*, необходимо использовать функцию *str\_replace*().

#### Отрезание пробелов.

Для отрезания пробелов используется фугкция *trim*().

string trim (string $st [, string $charlist])

*trim*() возвращает копию $*st*, только с удаленными ведущими и концевыми пробельными символами. Под пробельными символами здесь и далее подразумеваются:

* пробел " ";
* символ перевода строки \*n*;
* символ возврата каретки \*r*;
* символ табуляции \*t*;
* нулевой байт \0;
* вертикальная табуляция \*x0B*.

Установить альтернативный набор пробельных символов можно при помощи необязательного параметра $*charlist*. Он представляет собой строку, в которой перечислены все символы, подлежащие удалению.

Функция *ltrim*() делает то же, что и *trim*(), только удаляет исключительно ведущие пробелы, а концевые не трогает.

Функция *rtrim*() или *chop*() удаляет только концевые пробелы.

#### Изменение регистра

Функция *strtolower*() преобразует строку $*st* в нижний регистр. Возвращает результат перевода.

string strtolower(string $st)

Функция *strtoupper*() переводит строку $*st* в верхний регистр. Возвращает результат преобразования. Эта функция также работает со строками, в однобайтовой кодировке.

И наконец, функция *ucfirst*() преобразует в верхний регистр только первую букву в строке $*st*, не трогая остальные.

### 4.2.4. Работа с многобайтовыми строками

Для работы с многобайтовыми строками в *PHP* используется библиотека ***mbstring***. Она содержит аналоги обычных строковых функций с приставкой "*mb*\_", для поддержки работы с многобайтовыми строками.

Отметим, что не для всех функций имеются аналоги, а для некоторых случаев, создание такого аналога не требуется. Например, для функций замены или обрезки пробельных символов.

Рассмотрим несколько примеров работы с функциями из *mbstring*, в дальнейшем разделе приведена сводная таблица по имеющимся в ней функциям.

Пример 4.11. Различие между *mb\_strlen*() и *strlen*().

<?php

$str = "Привет, мир!";

echo strlen($str); // вернет 21

echo mb\_strlen($str); // вернет 12

В примере 4.11 стандартная функция *strlen*() для строки "Привет, мир!" вернула 21 байт. При этом 3 байта отводится под запятую, пробел и восклицательный знак, а оставшиеся 18 байт под 9 букв русского алфавита. Функция *mb\_strlen*() подсчитала количество символов в строке с учетом того, что под разные символы отводится разное количество байтов.

Поскольку для многобайтовой строки не работает индексирование, нельзя обратиться к букве по индексу. Для решения этой задачи придется использвоать *mb\_substr*().

Пример 4.12. Побуквенный перебор многобайтовой строки.

// Программа подсвечивает гласные в тексте красным

$text = "Я сегодня не работал и устал.";

$vowelsArr = ['у', 'е', 'ы', 'а', 'о', 'э', 'я', 'и', 'ю', 'ё'];

$html = "<style>b {color: red; font-weight: normal;}</style>\n";

//Последовательно выделяем буквы с помощью подрезки строк

for ($letterIndex = 0; $letterIndex < mb\_strlen($text); $letterIndex++) {

$letter = mb\_substr($text, $letterIndex, 1);

$html .= (in\_array(mb\_strtolower($letter), $vowelsArr))

? "<b>".$letter."</b>"

: $letter;

}

print $html;

Таким образом, посимвольная работа с многобайтовой строкой может быть довольно громоздкой. Для обработки строк целесообразнее использовать регулярные выражения.

### 4.2.5. Сводная таблица строковых функций

В таблице 4.1 представлены функции для работы с однобайтовыми и многобайтовыми строками. Функции для работы с многобайтовыми строками являются более универсальными (их можно использовать и для однобайтовых строк), поэтому целесообразно использовать именно их.

**Таблица 4.1.** Сводная таблица строковых функций.

|  |  |  |
| --- | --- | --- |
| Однобайтовые строки | Многобайтовые строки | Описание |
| *chr* | *mb\_chr* | Возвращает символ по коду. |
| *ord* | *mb\_ord* | Возвращает код по символу. |
| *strlen* | *mb\_strlen* | Возвращает длину строки. |
| *strcmp* | *strcmp* | Сравнение строк. |
| *strpos* | *mb\_strpos* | Позиция первого вхождения подстроки. |
| *stripos* | *mb\_stripos* | Позиция первого вхождения подстроки без учета регистра. |
| *strrpos* | *mb\_strrpos* | Позиция последнего вхождения подстроки. |
| *strpbrk* | *strpbrk* | Строка начиная с символа из заданного набора. |
| *substr* | *mb\_substr* | Возвращает подстроку. |
| *strtok* | *strtok* | Разбивает строку на токены. |
| **Однобайтовые строки** | **Многобайтовые строки** | **Описание** |
| *str\_replace* | *str\_replace* | Заменяет все вхождения строки поиска на строку замены. |
| *str\_ireplace* | *str\_ireplace* | Регистронезависимый вариант *str\_replace*. |
| *substr\_replace* | *substr\_replace* | Заменяет часть строки. |
| *str\_replace* | *str\_replace* | Заменяет все вхождения строки поиска на строку замены. |
| *strtr* | *mb\_strstr* | Преобразует заданные символы или заменяет подстроки. |
| *trim* | *trim* | Удаляет пробельные символы из начала и конца строки. |
| *ltrim* | *ltrim* | Удаляет пробельные символы из начала строки. |
| *rtrim* | *rtrim* | Удаляет пробельные символы с конца строки. |
| *strtolower* | *mb\_strtolower* | Преобразует строку в нижний регистр. |
| *strtoupper* | *mb\_strtoupper* | Преобразует строку в верхний регистр. |
| *ucfirst* | *ucfirst* | Преобразует первый символ строки в верхний регистр. |

# Глава 5. Математические функции

## 5.1. Округление чисел

Функция *abs*() возвращает модуль числа. Тип параметра $*number* может быть *float* или *int*, а тип возвращаемого значения всегда совпадает с типом этого параметра:

mixed abs(mixed $number)

Функция *round*() осуществляет математическое округление $*val*: числа с плавающей точкой округляются в сторону меньшего числа, если значение после запятой меньше 0.5, и в сторону большего числа, если значение после запятой больше или равно 0.5.

double round(double $val, [, int $precision = 0 [, int mode = PHP\_ROUND\_HALF\_UP]])

Второй необязательный параметр функции позволяет задать количество цифр после запятой, до которого осуществляется округление. Если параметр принимает отрицательное значение, округление осуществляется до позиции влево от запятой.

Пример 5.1. Использование функции *round*().

<?php

$foo = round(3.4); // $foo == 3.0

$foo = round(3.5); // $foo == 4.0

$foo = round(3.6); // $foo == 4.0

$foo = round(123.256, 2); // $foo = 123.26

$foo = round(127.256, -2); // $foo = 100.0

По умолчанию при использовании функции *round*() знак числа не имеет значения, учитывается только его абсолютная величина.

Изменить поведение *round*() можно при помощи третьего параметра $*mode*, который принимает одну из констант:

* *PHP\_ROUND\_HALF\_UP* – значения 0.5 округляются в большую сторону, т. е. 3.5 в 4, –3.5 в –4;
* *PHP\_ROUND\_HALF\_DOWN* – значения 0.5 округляются в меньшую сторону, т. е. 3.5 в 3, –3.5 в –3;
* *PHP\_ROUND\_HALF\_EVEN* – значения 0.5 округляются в сторону ближайшего четного числа, т. е. и 3.5, и 4.5 будет округлено до 4;
* *PHP\_ROUND\_HALF\_ODD* – значения 0.5 округляются в сторону ближайшего нечетного числа, т. е. 3.5 в 3, а 4.5 в 5.

Функция *ceil*() округляет аргумент $*number* всегда в сторону большего числа. Причем, в отличие от функции *round*(), при вычислении всегда учитывается знак $*number.*

int ceil(float $number)

Пример 5.2. Округление вверх.

<?php

$foo = ceil(3.1); // $foo == 4

$foo = ceil(3); // $foo == 3

Функция *floor*() округляет $*number* всегда в сторону меньшего числа. Так же, как и в случае *ceil*(), при округлении учитывается знак переданного числа.

int floor(float $number)

## 5.2. Генерация случайных чисел

При рассмотрении генераторов случайных чисел важна их равномерность. Хороший генератор не должен отдавать предпочтение цифре 7 или реже выбирать цифру 4.

Для генерации случайного числа в PHP можно использовать функции *rand*() и *mt\_rand*(). Начиная с *PHP 7.1.0*, *rand*() использует тот же алгоритм получения случайных чисел, что и *mt\_rand*(), поэтому, в настоящее время эти функции идентичны.

int rand (int $min, int $max)

Пример 5.3. Извлечение случайной строки из файла.

<?php

$file = file("largetextfile.txt");

echo $file[rand(0, count($file) - 1)];

Для задания начального значения генератора случайных чисел на новую последовательность, можно использовать функцию *mt\_srand*().

void mt\_srand(int $seed)

Пример 5.4. Последовательности случайных чисел.

<?php

mt\_srand(123);

for ($i = 0; $i < 5; $i++) {

echo mt\_rand()." ";

echo "<br />";

}

//Будут сгенерированы те же числа

mt\_srand(123);

for ($i = 0; $i < 5; $i++) {

echo mt\_rand()." ";

}

Обе серии случайных чисел, которые напечатает скрипт, будут совпадать. Перед выборкой очередных чисел функцией *mt\_rand*() установлена одна и та же последовательность случайных чисел (ее "ключ" – 123). Таким образом, числа, выдаваемые *mt\_rand*(), не являются случайными. Зная первое число в цепочке и идентификатор последовательности, всегда можно предсказать, какими будут следующие величины. Именно поэтому числа, которые генерирует функция *mt\_rand*(), называют псевдослучайными.

В современных версиях *PHP* первый вызов *mt\_rand*() автоматически и неявно вызывает *mt\_srand*(), так что в большинстве ситуаций необходимости применять *mt\_srand*() в явном виде нет.

Начиная с *PHP* 7*,* доступна еще одна реализация генератора псевдослучайных чисел с хорошей равномерностью. Она реализована в функции *random\_int*().

int random\_int(int $min = PHP\_INT\_MIN, int $max = PHP\_INT\_MAX)

## 5.3. Преобразования между системами счисления

Основной функции преобразования числа в другие системы счисления является *base\_convert*().

string base\_convert(string $number, int $frombase, int $tobase)

*base\_convert*() переводит число $*number*, заданное как строка в системе счисления по основанию $*frombase*, в систему по основанию $*tobase*. Параметры $*frombase* и $*tobase* могут принимать значения только от 2 до 36 включительно. В строке $*number* цифры обозначают сами себя, буква a соответствует 11, *b* – 12 и т. д. до *z*, которая обозначает 36.

Пример 5.5. Применение функции *base*\_*convert*().

<?php

echo base\_convert("FF", 16, 2); // 11111111

Для преобразования в десятичную систему можно использовать следующие функции:

int bindec(string $num\_string)

int octdec(string $num\_string)

int hexdec(string $num\_string)

Эти функции преобразуют двоичное (функция *bindec*), восьмеричное (*octdec*) или шестнадцатеричное (*hexdec*) число, заданное в строке $*num\_string*, в десятичное число.

Для обратного преобразования можно использовать функции:

string decbin(int $number)

string decoct(int $number)

string dechex(int $number)

Они возвращают строку, представляющую собой двоичное (соответственно, восьмеричное или шестнадцатеричное) представление целого числа $*number*. Максимальное число, которое еще может быть преобразовано, равно 2 147 483 647.

## 5.4. Поиск минимума и максимума

Рассмотрим функцию *min*().

mixed min(mixed $arg1 [mixed $arg2, ..., mixed $argn])

Данная функция возвращает наименьшее из чисел, заданных в ее аргументах. Различают два способа вызова этой функции: с одним параметром или с несколькими. Если указан лишь один параметр (первый), то он обязательно должен быть массивом.

В этом случае возвращается минимальный элемент массива. В противном случае первый и остальные аргументы трактуются как числа с плавающей точкой, они сравниваются, и возвращается наименьшее.

Тип возвращаемого значения выбирается так: если хотя бы одно из чисел, переданных на вход, задано в формате с плавающей точкой, то и результат будет с плавающей точкой, в противном случае результат будет целым числом. С помощью этой функции нельзя сравнивать строки.

Функция *max*() работает аналогично *min*(), только ищет максимальное значение, а не минимальное.

mixed max(mixed $arg1 [mixed $arg2, ..., mixed $argn])

## 5.5. Не-числа

Некоторые операции, такие как извлечение корня или возведение в дробную степень, нельзя выполнять с отрицательными числами. При попытке выполнения такого действия функции вместо результата возвращают специальные "псевдочисла" – *NAN* (не-число), +*Infinite* (+ ∞) или – *Infinite* (– ∞).

Для распознавания не-чисел можно использовать функцию *is\_nan*().

bool is\_nan(mixed $variable)

Функция возвращает для *NAN* значение *true*, а для всех остальных чисел (в том числе и для бесконечностей) – *false*. При помощи этой функции можно отличить *NAN* от любого другого числа.

Чтобы определить, является ли число конечным, можно использовать функцию *is\_infinite*().

bool is\_infinite(mixed $variable)

Бесконечность возникает,например, при возведении 0 в отрицательную степень: *pow*(0, -1) вернет +*Infinite*.

С *NAN* можно выполнять арифметические операции. Например, после выполнеия команды:

echo 1/pow(0, -1)

Будет напечатан 0 так, как 1 поделить на бесконечность равно нулю.

Бесконечность может быть с плюсом и минусом. В первом случае ее строковым представлением является 1. #*INF*, а во втором – значение -1.#*INF*.

Начиная с версии *PHP* 7, деление на ноль дает бесконечность. При этом для совместимости с предыдущими версиями предупреждение о делении на ноль все равно выдается.

## 5.6. Степенные функции

Перечислим доступные в PHP степенные функции.

float sqrt(float $arg)

Возвращает квадратный корень из аргумента. Если аргумент отрицателен, возвращается специальное "псевдочисло" *NAN*, работа программы не прекращается.

float log(float $arg)

Возвращает натуральный логарифм аргумента. В случае недопустимого числа может вернуть +*Infinite*, –*Infinite* или *NaN*.

float exp(float $arg)

Возвращает *e* (2,718281828...) в степени $*arg*.

float pow(float $base, float $exp)

Возвращает $*base* в степени $*exp*. Может вернуть +*Infinite*, –*Infinite* или *NAN* в случае недопустимых аргументов.

## 5.7. Тригонометрические функции

В тригонометрических выражениях большое значение играет число π, которое равно половине длины окружности с единичным радиусом.

Для получения числа π в *PHP*-сценариях предназначена специальная функция.

double pi()

Помимо функции *pi*() можно воспользоваться константой *M\_PI*.

Рассмотрим основные тригонометрические функции *PHP*.

float deg2rad(float $deg)

Переводит градусы в радианы, т. е. возвращает число $*deg*/180\**M\_PI*. Все тригонометрические функции в *PHP* работают именно с радианами, но не с градусами.

float rad2deg(float $deg)

Наоборот, переводит радианы в градусы.

float acos(float $arg)

Возвращает арккосинус аргумента.

float asin(float $arg)

Возвращает арксинус аргумента.

float atan(float $arg)

Возвращает арктангенс аргумента.

float atan2(float $y, float $x)

Возвращает арктангенс величины $*y*/$*x*, но с учетом той четверти, в которой лежит точка ($*x*, $*y*). Эта функция возвращает результат в радианах, принадлежащий отрезку от −∞ до +∞.

float sin(float $radians)

Возвращает синус аргумента. Аргумент задается в радианах.

float cos(float $radians)

Возвращает косинус аргумента.

float tan(float $radians)

Возвращает тангенс аргумента, заданного в радианах.

## 5.8. Сводная таблица математических функций

В таблице 5.1 кратко перечислены основные математические функции *PHP*.

**Таблица 4.2.** Сводная таблица математических функций.

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| **Функции округления** | |
| *abs* | Возвращает абсолютное значение числа |
| *round* | Округляет число типа *float* |
| *ceil* | Округляет дробь в большую сторону |
| *floor* | Округляет дробь в меньшую сторону |
| **Случайные числа** | |
| *rand* | Генерирует случайное число |
| *mt\_rand* | Генерирует случайное значение методом с помощью генератора простых чисел на базе Вихря Мерсенна |
| *mt\_getrandmax* | Показывает максимально возможное значение случайного числа |
| *mt\_srand* | Переинициализирует генератор случайных чисел |
| *random\_int* | Генерирует криптографически безопасные псевдослучайные целые числа |
| **Системы счисления** | |
| *base\_convert* | Преобразует число между произвольными системами счисления |
| *bindec* | Двоичное в десятичное |
| *octdec* | Восьмеричное в десятичное |
| *hexdec* | Шестнадцатеричное в десятичное |
| *decbin* | Десятичное в двоичное |
| *decoct* | Десятичное в восьмеричное |
| *dechex* | Десятичное в шестнадцатеричное |
| **Минимум и максимум** | |
| *min* | Наименьшее значение |
| *max* | Наибольшее значение |
| **Не-числа** | |
| *is\_nan* | Является ли значение "не числом" |
| *is\_infinite* | Является ли значение бесконечным |
| **Степенные функции** | |
| *sqrt* | Квадратный корень |
| *log* | Натуральный логарифм |
| *exp* | Степень числа *е* |
| *pow* | Возведение в степень |
| **Тригонометрические функции** | |
| *pi* | Число *π* |
| *deg2rad* | Преобразует значение из градусов в радианы |
| *rad2deg* | Преобразует значение из радиан в градусы |
| *acos* | Арккосинус |
| *asin* | Арксинус |
| *atan* | Арктангенс |
| *atan2* | Арктангенс двух переменных |
| *sin* | Синус |
| *cos* | Косинус |
| *tan* | Тангенс |

# Глава 6. Работа с файловой системой

## 6.1. Работа с файлами и каталогами

### 6.1.1. Открытие файла

Работа с файлами в *PHP* разделяется на три этапа. Сначала файл открывается в нужном режиме, при этом возвращается некое целое число, служащее идентификатором открытого файла (дескриптор файла). Затем настает очередь команд работы с файлом, эти команды привязаны к дескриптору файла, а не к его имени. После этого файл следует закрыть. *PHP* автоматически закроет все файлы по завершении сценария.

Для открытия файла используется функция *fopen*().

int fopen(string $file, string $mode, bool $use\_include=false, resource $context)

*fopen*() открывает файл с именем *$file* в режиме *$mode* и возвращает дескриптор открытого файла. Если файл открыть не удалось, то *fopen*() возвращает *false*. Необязательный параметр *$use\_include* сообщает *PHP* о том, что, если задано относительное имя файла, его следует искать также и в списке путей, используемом инструкциями *include* и *require*.

Параметр *$mode* может принимать значения, приведенные в первом столбце таблицы 6.1

**Таблица 6.1.** Режимы открытия файла функцией *fopen*().

| **Режим** | **Описание** |
| --- | --- |
| r | Открывает файл только для чтения; помещает указатель в начало файла. |
| r+ | Открывает файл для чтения и записи; помещает указатель в начало файла. |
| w | Открывает файл только для записи; помещает указатель в начало файла и обрезает файл до нулевой длины. Если файл не существует - пробует его создать. |
| w+ | Открывает файл для чтения и записи; помещает указатель в начало файла и обрезает файл до нулевой длины. Если файл не существует - пытается его создать. |
| a | Открывает файл только для записи; помещает указатель в конец файла. Если файл не существует - пытается его создать. В данном режиме функция [fseek()](https://www.php.net/manual/ru/function.fseek.php) не применима, записи всегда добавляются в конец. |
| a+ | Открывает файл для чтения и записи; помещает указатель в конец файла. Если файл не существует - пытается его создать. В данном режиме функция [fseek()](https://www.php.net/manual/ru/function.fseek.php) влияет только на позицию чтения, записи всегда добавляются в конец. |
| x | Создаёт и открывает только для записи; помещает указатель в начало файла. Если файл уже существует, вызов **fopen()** закончится неудачей, вернёт FALSE и выдаст ошибку уровня E\_WARNING. Если файл не существует, попытается его создать. Это эквивалентно указанию флагов O\_EXCL|O\_CREAT для внутреннего системного вызова open(2). |
| x+ | Создаёт и открывает для чтения и записи; в остальном имеет то же поведение, что и'x'. |
| c | Открывает файл только для записи. Если файл не существует, то он создается. Если же файл существует, то он не обрезается (в отличие от 'w'), и вызов к этой функции не вызывает ошибку (также как и в случае с 'x'). Указатель на файл будет установлен на начало файла. Это может быть полезно при желании заблокировать файл (см. [flock()](https://www.php.net/manual/ru/function.flock.php)) перед изменением, так как использование 'w' может обрезать файл еще до того как была получена блокировка (если вы желаете обрезать файл, можно использовать функцию [ftruncate()](https://www.php.net/manual/ru/function.ftruncate.php) после запроса на блокировку). |
| c+ | Открывает файл для чтения и записи; в остальном имеет то же поведение, что и 'c'. |

Но это еще не полное описание параметра *$mode*. В конце любой из строк *r, w, a, x, c, r+, w+, a+ x+* и *c+* может находиться еще один необязательный символ – *b* или *t*. Если указан *b*, то файл открывается в режиме бинарного чтения/записи. Если же это *t*, то файл воспринимается как текстовый.

Последний параметр *$context* позволяет задать контекст потока в сетевых операциях, например, настроить значение *USER\_AGENT* при обращении к файлу, расположенному на *http-* или *ftp-*серверах.

Пример 6.1. Использование функции *fopen*().

// Открывает файл на чтение

$f = fopen("/home/user/file.txt", "rt") or die("Ошибка!");

// Открывает HTTP-соединение на чтение

$f = fopen("http://www.php.net/", "rt") or die("Ошибка!");

// Открывает FTP-соединение с указанием имени входа и пароля для записи

$f = fopen("ftp://user:pass@example.com/a.txt", "wt") or die("Ошибка!");

Не стоит использовать обратные слеши (\) в именах файлов, как это принято в *DOS* и *Windows*. *PHP* в нужный момент переводит прямые слеши (/) в обратные. Если нельзя обойтись без обратного слеша, его необходимо будет удвоить, потому что в строках он воспринимается как спецсимвол.

Обратные слеши особенно коварны тем, что иногда можно забыть их удвоить, и программа, казалось бы, по-прежнему будет работать.

Пример 6.2. Проблема обратных слешей.

<?php

$path = "c:\non\existent\file";

//Ошибка!

//Путь равен c: on\existent\file

echo $path."<br />";

Во строке пропала одна буква и добавился пробел. В действительности там, конечно же, не пробел, а символ перевода строки. Сочетание *\n* в строке заменилось единственным байтом – символом конца строки, в то время как *\e* и *\f*, *\w* и т. д. остались сами по себе – ведь это не специальные комбинации.

### 6.1.2. Закрытие файла

После работы файл лучше всего закрыть с помощью функции *fclose*(), хоть это делается и автоматически при завершении сценария.

int fclose(int $fp)

Функция закрывает файл, открытый предварительно функцией *fopen*(). Возвращает *false*, если файл закрыть не удалось (например, что-то с ним случилось или же разорвалась связь с удаленным хостом). В противном случае возвращает *true*.

Необходимо всегда закрывать *FTP*- и *HTTP*-соединения, таким образом вы уменьшите нагрузку на сервер и будете уверены в том, что все данные были доставлены без ошибок.

### 6.1.3. Чтение и запись

Для каждого открытого файла система хранит определенную величину, которая называется текущей позицией ввода/вывода, или указателем файла. Функции чтения и записи файлов читают блок данных, начиная с этой позиции, а функции записи – записывают, также отсчитывая от нее. Если указатель файла установлен за последним байтом и осуществляется запись, то файл автоматически увеличивается в размере. Есть также функции для установки этой самой позиции в любое место файла.

#### Блочные чтения и запись

Рассмотрим функции блочного чтения и записи.

string fread(int $f, int $numbytes)

Функция *fread*() читает из файла *$f* блок из *$numbytes* символов и возвращает строку этих символов. После чтения указатель файла продвигается к следующим после прочитанного блока позициям. Разумеется, если *$numbytes* больше, чем можно прочитать из файла, возвращается то, что удалось считать. Этот прием можно использовать, если нужно считать в строку файл целиком. Для этого просто задайте в *$numbytes* очень большое число. Но если вы заботитесь об экономии памяти в системе, так поступать не рекомендуется. Дело в том, что в некоторых версиях *PHP* передача большой длины строки во втором параметре *fread*()вызывает первоначальное выделение этой памяти в соответствии с запросом (даже если строка гораздо короче).

int fwrite(int $f, string $st [, int $length]) )

Функция *fwrite*() записывает в файл *$f* все содержимое строки *$st*. Если указан необязательный параметр *$length*, запись в файл прекращается либо по достижению конца строки $st, либо при записи *$length* байт, в зависимости от того, что произойдет раньше.

При работе с текстовыми файлами (т. е. когда указан символ t в режиме открытия файла) все \n автоматически преобразуются в тот разделитель строк, который принят в вашей операционной системе.

#### Построчные чтения и запись

Для чтения строки из файла можно воспользоваться функцией *fgets*().

string fgets(int $f [, int $length])

Если строка в файле занимает больше *$length-1* байтов, то возвращаются только ее *$length-1* символов. Для построчной записи можно использовать функцию *fputs*().

#### Чтение и запись целого файла

Для построчного считывания файла в список подойдет функция *file*().

list file(string $filename [, int $flags [, resource $context]])

Она считывает файл с именем *$filename* целиком (в бинарном режиме) и возвращает массив-список, каждый элемент которого соответствует строке в прочитанном файле.

**Таблица 6.2.** Константы параметра *$flags* функции *file*()*.*

|  |  |
| --- | --- |
| Параметр | Описание |
| *FILE\_USE\_INCLUDE\_PATH* | Осуществлять поиск в каталогах библиотек PHP |
| *FILE\_IGNORE\_NEW\_LINES* | Не добавлять символ новой строки \n в конец каждого элемента массива |
| *FILE\_SKIP\_EMPTY\_LINES* | Пропускать пустые строки |

При необходимости указать более одной константы, их следует объединить при помощи побитового оператора "|".

Функция *file\_get\_contents*() считывает целиком файл *$filename* и возвращает все его содержимое в виде одной единственной строки.

string file\_get\_contents(string $filename, bool $use\_include\_path = false, resource $context, int $offset = -1, int $maxlen)

Параметр *$offset* позволяет задать смещение в байтах, начиная с которого осуществляется чтение содержимого *$offset*. Данный параметр не работает для сетевых обращений. Последний параметр *$maxlen* позволяет задать максимальный размер считываемых данных.

Функция *file\_put\_contents*() позволяет в одно действие записать данные *$data* в файл, имя которого передано в параметре *$filename*. При этом данные записываются, как есть - трансляция переводов строк не производится.

string file\_put\_contents(string $filename, string $data, bool $flags = 0, resource $context)

Пример 6.3. Чтение и запись целого файла.

$data = file\_get\_contents("image.gif");

file\_put\_contents("newimage.gif", $data);

**Таблица 6.3.** Константы параметра *$flags* функции *file\_put\_contents*()*.*

|  |  |
| --- | --- |
| Параметр | Описание |
| *FILE\_APPEND* | Производит дописывание в конец файла |
| *FILE\_USE\_INCLUDE\_PATH* | Находит файл в путях поиска библиотек, используемых функциями include() и require() |
| *LOCK\_EX* | Функция получает эксклюзивную блокировку файла на время записи |

### 6.1.4. Работа с путями

Довольно часто приходится сталкиваться с обработкой путей к файлам. Например, добавлять к имени слева путь к какому-то каталогу или, наоборот, из полного пути выделить непосредственное имя файла.

Для выделения имени файла, можно воспользоваться функцией *basename*().

string basename(string $path, string $suffix)

Пример 6.4. Выделяет имя файла из полного пути.

<?php

echo basename("/home/somebody/somefile.txt"); // выводит "somefile.txt"

Если указан необязательный параметр *$suffix* и имя файла заканчивается на содержимое этой строки, оно будет отброшено. Этот параметр удобно использовать для вырезания расширения файла.

Пример 6.5. Использование функции *basename*().

// выводит "somefile"

echo basename("/home/somebody/somefile.txt", ".txt");

Функция *dirname*() возвращает имя каталога, выделенное из пути *$path*.

string dirname(string $path, int $levels = 1)

Пример 6.6. Применение функции *dirname*().

// выводит "/home"

echo dirname("/home/file.txt");

// выводит ".."

echo dirname("../file.txt");

// выводит "/" под UNIX, "\" под Windows

echo dirname("/file.txt");

// то же самое

echo dirname("/");

// выводит "."

echo dirname("file.txt");

Параметр *$levels*, появившийся в *PHP* 7, позволяет задать уровень извлекаемого родительского каталога. По умолчанию параметр принимает значение 1, однако, изменив значение на 2 или 3, можно извлекать родительские каталоги более высоких уровней

Пример 6.7. Извлечение родительских каталогов.

<?php

# /usr/opt/local/etc/

echo dirname("/usr/opt/local/etc/hosts");

# /usr/opt/local

echo dirname("/usr/opt/local/etc/hosts", 2);

# /usr/opt

echo dirname("/usr/opt/local/etc/hosts", 3);

Функция *tempnam*() генерирует имя файла в каталоге *$dir* с префиксом *$prefix* в имени, причем так, чтобы созданный под этим именем в будущем файл был уникален.

string tempnam(string $dir, string $prefix)

Для этого к строке *$prefi*x присоединяется некое случайное число. Например, вызов *tempnam("/tmp","temp")* может возвратить что-то типа */tmp/temp3a6b243c*.

Функция realpath() преобразует относительный путь *$path* в абсолютный, т. е. начинающийся от корня.

string realpath(string $path)

Пример 6.8. Применение функции *realpath*().

// абсолютный путь, например, /home/t.php

echo realpath("../t.php");

// выводит имя текущего каталога

echo realpath(".");

### 6.1.5. Работа с файлами

Функции, представленные в этом разделе, упрощают работу с целыми файлами, когда нет необходимости читать их построчно или поблочно.

bool copy(string $src, string $dst, resource $context)

Копирует файл с именем *$src* в файл с именем *$dst*. При этом, если файл *$dst* на момент вызова существовал, выполняется его перезапись. Функция возвращает *true*, если копирование прошло успешно, а в случае провала – *false.*

bool rename(string $oldname, string $newname [, resource $context])

Переименовывает (или перемещает, что одно и то же) файл с именем *$oldname* в файл именем *$newname*. Если файл *$newname* уже существует, регистрируется ошибка, и функция возвращает *false*. То же происходит и при прочих неудачах. Если же все прошло успешно, возвращается *true*.

bool unlink(string $filename [, resource $context])

Удаляет файл с именем *$filename*. В случае неудачи возвращает *false*, иначе – *true*.

### 6.1.6. Работа с каталогами

Создание каталога осуществляется с помощью функции *mkdir*().

bool mkdir(string $name, int $perms = 0777, bool $recursive = false, resource $context)

*mkdir*() создает каталог с именем *$name* и правами доступа *$perms*. Права доступа для каталогов указываются точно так же, как и для файлов. Чаще всего значение *$perms* устанавливают равным 0770 (предваряющий ноль указывает на то, что это восьмеричная константа). Необязательный параметр *$recursive* позволяет создавать отсутствующие в пути каталоги одним вызовом. В случае успеха функция возвращает *true*, иначе – *false*. Необходимо заметить, что пользователь не может создать подкаталог в родительском каталоге, права на запись в который у него отсутствуют.

Функция *rmdir*() удаляет каталог с именем *$name*.

bool rmdir(string $name [, resource $context])

В случае успеха возвращает *true*, иначе – *false*.

Функция *chdir*() сменяет текущий каталог на указанный.

bool chdir(string $path)

Если такой каталог не существует, возвращает *false*. Параметр *$path* может определять и относительный путь, задающийся от текущего каталога

Функция *getcwd*() возвращает полный путь к текущему каталогу, начиная от корня (/).

#### Чтение каталогов

Дальше описываются функции, которые позволяют узнать, какие объекты находятся в указанном каталоге. Механизм работы этих функций базируется примерно на тех же принципах, что и для файловых операций: сначала интересующий каталог открывается, затем из него производится считывание записей, и под конец каталог нужно закрыть.

int opendir(string $path [, resource $context])

Открывает каталог *$path* для дальнейшего считывания из него информации о файлах и подкаталогах и возвращает его идентификатор. Дальнейшие вызовы *readdir*() с идентификатором в параметрах будут обращены именно к этому каталогу. Функция возвращает *false*, если произошла ошибка.

string readdir(int $handle)

Считывает очередное имя файла или подкаталога из открытого ранее каталога с идентификатором *$handle* и возвращает его в виде строки. Порядок следования файлов в каталоге зависит от операционной системы – скорее всего, он будет совпадать с тем порядком, в котором эти файлы создавались, но не всегда. Вместе с именами подкаталогов и файлов будут также получены два специальных элемента: это "." (ссылка на текущий каталог) и ".." (ссылка на родительский каталог).

void closedir(int $handle)

Закрывает ранее открытый каталог с идентификатором *$handle*. Не возвращает ничего.

void rewinddir(int $handle)

"Перематывает" внутренний указатель открытого каталога на начало. После этого можно воспользоваться *readdir*()*,* чтобы заново начать считывать содержимое каталога.

#### Чтение каталога с помощью *glob*()

Гораздо удобнее производить чтение каталога с помощью функции *glob*().

list glob(string $pattern [, int $flags])

Функция возвращает список всех путей, которые подходят под маску *$pattern*. Например, вызов *glob("\*.txt")* вернет список всех текстовых файлов в текущем каталоге, а *glob("c:/windows/\*.exe")* – всех *exe*-файлов в каталоге windows (вместе с путями). Метасимвол '\*' означает любое число любых символов, тогда как '?' – один любой символ.

Пример 6.9. Обработка содержимого каталога с помощью *glob*().

<?php

foreach(glob("./\*") as $fileName)

if(is\_file($fileName))

print $fileName."<br/>";

Значения параметра $*flags* функции glob() представлены в таблице 6.4.

**Таблица 6.4.** Константы параметра $*flags* функции *glob*().

|  |  |
| --- | --- |
| Параметр | Описание |
| *GLOB\_ONLYDIR* | В результирующий список попадут только имена каталогов, но не файлов |
| *GLOB\_BRACE* | Позволяет задавать альтернативы в выражении, перечисляя их через запятую в фигурных скобках. Например, glob("c:/windows/{\*.exe,\*.ini}" |
| *GLOG\_ERR* | Остановить работу функции при ошибке чтения, по умолчанию такие ошибки игнорируются |
| *GLOB\_MARK* | Добавляет слеш (\ в Windows, / в UNIX) к тем элементам результирующего списка, которые являются каталогами |
| *GLOB\_NOSORT* | По умолчанию результирующий массив сортируется по алфавиту. Данный флаг запрещает это делать, что может немного улучшить производительность программы. Если он установлен, элементы в списке будут идти в том же порядке, в каком они записаны в каталоге |
| *GLOB\_NOCHECK* | В случае, если под маску не подошел ни один файл или каталог, функция вернет список из единственного элемента, равного $pattern |
| *GLOB\_NOESCAPE* | Имена файлов в UNIX могут содержать служебные символы вроде звездочки (\*), вопросительного знака (?) и т. д. Если флаг GLOB\_NOESCAPE не указан, функция вернет их в списке, предварив все специальные символы обратными слешами (\). Если же флаг указан, имена возвращаются, как есть. |

### 6.1.7. Перечень функций для работы с файловой системой

**Таблица 6.5.** Функции для работы с файловой системой.

|  |  |
| --- | --- |
| Название | Описание |
| *basename* | Возвращает последний компонент имени из указанного пути |
| *chgrp* | Изменяет группу файла |
| *chmod* | Изменяет режим доступа к файл |
| *chown* | Изменяет владельца файла |
| *copy* | Копирует файл |
| *delete* | См. описание функции unlink или unset |
| *dirname* | Возвращает имя родительского каталога из указанного пути |
| *fclose* | Закрывает открытый дескриптор файла |
| *feof* | Проверяет, достигнут ли конец файла |
| *fgetc* | Считывает символ из файла |
| *fgetcsv* | Читает строку из файла и производит разбор данных CSV |
| **Название** | **Описание** |
| *fgets* | Читает строку из файла |
| *file\_exists* | Проверяет существование указанного файла или каталога |
| *file\_get\_contents* | Читает содержимое файла в строку |
| *file\_put\_contents* | Пишет строку в файл |
| *file* | Читает содержимое файла и помещает его в массив |
| *fileatime* | Возвращает время последнего доступа к файлу |
| *filectime* | Возвращает время изменения индексного дескриптора файла |
| *filegroup* | Получает идентификатор группы файла |
| *fileinode* | Возвращает индексный дескриптор файла |
| *filemtime* | Возвращает время последнего изменения файла |
| *fileowner* | Возвращает идентификатор владельца файла |
| *fileperms* | Возвращает информацию о правах на файл |
| *filesize* | Возвращает размер файла |
| *filetype* | Возвращает тип файла |
| *flock* | Портируемая консультативная блокировка файлов |
| *fopen* | Открывает файл или URL |
| *fread* | Бинарно-безопасное чтение файла |
| *fscanf* | Обрабатывает данные из файла в соответствии с форматом |
| *fseek* | Устанавливает смещение в файловом указателе |
| *unlink* | Удаляет файл |
| *touch* | Устанавливает время доступа и модификации файла |
| *tmpfile* | Создаёт временный файл |
| *stat* | Возвращает информацию о файле |
| *fwrite* | Бинарно-безопасная запись в файл |
| *is\_dir* | Определяет, является ли имя файла директорией |
| *is\_executable* | Определяет, является ли файл исполняемым |
| *is\_file* | Определяет, является ли файл обычным файлом |
| *is\_link* | Определяет, является ли файл символической ссылкой |
| *is\_readable* | Определяет существование файла и доступен ли он для чтения |
| *is\_uploaded\_file* | Определяет, был ли файл загружен при помощи HTTP POST |
| *is\_writable* | Определяет, доступен ли файл для записи |
| *is\_writeable* | Псевдоним is\_writable |
| **Название** | **Описание** |
| *readfile* | Выводит файл |
| *rename* | Переименовывает файл или директорию |
| *rmdir* | Удаляет директорию |

## 6.2. Блокировки

### 6.2.1. Понятие блокировки

При интенсивном обмене данными с файлами в многопользовательских системах встает вопрос синхронизации операций чтения/записи данных. Например, пусть есть несколько задач, изменяющих файл одновременно. Необходимо, чтобы в единицу времени к файлу имела доступ лишь одна задача, а остальные ожидали своей очереди. Это нужно, чтобы данные от нескольких задач не затирали друг друга.

В этом случае стоит воспользоваться функцией *flock*()*,* которая устанавливает так называемую "рекомендательную блокировку" (*advisory* *locking*) для файла. Это означает, что блокирование доступа осуществляется не на уровне ядра системы, а на уровне программы. Поясним на примере. Довольно известно сравнение рекомендательной блокировки с перекрестком, на котором движение регулируется светофором. Когда горит красный, одни машины стоят, а другие проезжают. В принципе, любая машина может поехать на красный, но в таком случае возможны аварии. Рекомендательная блокировка работает точно таким же образом. А именно задачи, работающие с ней должны работать правильно. Отметим, что в *Windows* ***flock*()** использует обязательную (*mandatory*) блокировку вместо рекомендательной.

### 6.2.2. Функция *flock*()

Единственная функция, которая занимается управлением блокировками в *PHP*, называется *flock*()*.*

bool flock(int $fp, int $type [, int& $wouldblock])

Функция устанавливает для указанного открытого дескриптора файла *$fp* режим блокировки, который бы хотел получить текущий процесс. Этот режим задается аргументом *$type* и может быть одной из следующих констант.

**Таблица 6.6.** Константы параметра $*type* функции *flock*().

|  |  |
| --- | --- |
| Параметр | Описание |
| *LOCK\_SH* | Разделяемая блокировка |
| *LOCK\_EX* | Исключительная блокировка |
| *LOCK\_UN* | Снять блокировку |
| *LOCK\_NB* | Эту константу нужно прибавить к одной из предыдущих, если вы не хотите, чтобы программа "подвисала" на flock() в ожидании своей очереди, а сразу возвращала управление. |

В случае если был затребован режим без ожидания, и блокировка не была успешно установлена, в необязательный параметр-переменную *$wouldblock* будет записано значение *true*. При ошибке функция возвращает *false*, а в случае успешного завершения – *true*.

Пример 6.10. Использование функции *flock*().

<?php

$fp = fopen("lock.txt", "r+");

// берем эксклюзивную блокировку

if (flock($fp, LOCK\_EX | LOCK\_NB)) {

ftruncate($fp, 0); // очищаем файл

fwrite($fp, "Что-нибудь пишем сюда\n");

fflush($fp); // сбрасываем буфер в файл

flock($fp, LOCK\_UN); // снимаем блокировку

}

else

echo "Не удалось получить блокировку!";

### 6.2.3. Типы блокировок

Как мы уже видели в таблице 6.6, существуют два основных типа блокировок – исключительная и разделяемая. Рассмотрим их подробнее.

#### Исключительная блокировка

Исключительная блокировка используется для того, чтобы ограничить доступ к файлу всем другим задачам, кроме текущей. Обычно такая блокировка устанавливается для обеспечения целостной записи в файл. То есть, пока наша задача пишет данные в файл – остальные задачи ждут.

Рассмотрим пример.

Пример 6.11. Счетчик посещений.

<?php

$counterFile = "counter.txt";

//создаем файл

if(!is\_file($counterFile)) touch($counterFile);

//открываем файл для записи, но не очищаем!

$fp = fopen($counterFile, "r+");

//берем исключительную блокировку

flock($fp, LOCK\_EX);

//вычисляем посещения

$data = intval(fread($fp, 100)); //или filesize();

$data++;

//пишем в файл

fseek($fp, 0); // перематываем указатель на начало

ftruncate($fp, 0); //подрезаем файл

fwrite($fp, $data);

//снимаем блокировку и закрываем файл

flock($fp, LOCK\_UN);

fclose($fp)

В данном примере мы блокируем файл полностью, для внесения изменений. Однако, если каждый раз при обращении к файлу блокировать файл исключительно, то производительность работы с файлом существенно снизится.

#### Разделяемая блокировка

Обычно с файлами работают как на запись так и на чтение. Причем, обычно операций чтения гораздо больше, чем операций записи. Например, если мы реализуем гостевую книгу, то пользователи чаще будут ее листать, нежели оставлять сообщения. В связи с этим, встает задача обеспечения эффективного чтения.

Как было описано выше, для избежания чтения неполных (изменяющихся) данных можно использовать исключительную блокировку, но такой подход приведет к тому, что пользователи будут читать файл строго по очереди, а это медленно. Именно для решения этой задачи нужна разделяемая блокировка.

Разделяемую блокировку могут взять несколько задач одновременно, тем самым обеспечивая параллельную работу с файлом. При этом, подразумевается, что задачи разумные и понимают, что в таком режиме вносить изменения в файл нельзя.

Таким образом, общая схемы работы выглядит так:

* если задаче нужно изменить данные в файле – она берет исключительную блокировку, при этом *flock*() дождется, пока все другие разделяемые и исключительные блокировки будут закрыты и даст задаче монопольный доступ к файлу;
* если задаче нужно читать файл без ошибок, то она запрашивает разделяемую блокировку, *flock*() даст ее, как только будут сняты все исключительные блокировки (при этом разделяемые блокировки могут существовать);
* не забудьте, что блокировка мешает окружающим обращаться к файлу, так что снимите ее так быстро, как сможете.
* обратите внимание, что при изменении данных исключительная блокировка должна быть взята ***до момента чтения данных***, подлежащих изменению (см. пример 6.10), в противном случае может возникнуть ситуация одновременного изменения данных и часть данных будет потеряна.

## 6.3. Управление правами доступа

### 6.3.1. Представление прав доступа

Когда вы регистрируетесь в системе (например, по *SSH* или *FTP*), вашей сессии присваивается так называемый идентификатор пользователя (*user ID, UID*). По идентификатору пользователя система определяет, какие действия с какими объектами ему выполнять разрешено, а какие следует запретить.

Идентификаторы пользователей могут для удобства объединяться в так называемые группы. Группа в *UNIX* – это просто список *UID*, имеющий свой собственный идентификатор группы (*group ID, GID*). Как и для *UID*, для *GID* может существовать легко запоминающееся буквенное имя группы. Каждому пользователю обязательно должна быть назначена какая-нибудь группа, которой он принадлежит.

Кроме идентификаторов владельца и группы, каждый файл имеет так называемые права доступа (*access* *permissions*). Права доступа определяют, какие действия разрешено выполнять с файлом (владельцу, группе владельца, другим пользователям):

* чтение (*r*, *read*);
* запись (*w*, *write*);
* исполнение (*x*, *eXecute*).

Итак, для каждого файла существует 9 флагов, объединенных в группу по три.

**Таблица 6.7.** Представление флагов прав доступа.

|  |  |  |
| --- | --- | --- |
| *USER* | *GROUP* | *OTHER* |
| *rwx* | *rwx* | *rwx* |

Или, более коротко: *rwxrwxrwx.*

Если какой-то из флагов не установлен (например, "остальным" пользователям запрещена запись в файл), то вместо него отображают прочерк: *rwxrwxr-x.*

Точно так же, как для имени пользователя и группы существует числовой идентификатор (*UID* или *GID*), для прав доступа имеется числовое представление. Обычно права рассматривают как набор из девяти битовых флагов, каждый из которых может принимать значение 0 или 1. Чтобы подчеркнуть логическое разбиение этих флагов на тройки битов, каждую из этих троек представляют одним числом от 0 до 7:

--- – 0

r-- – 4 (= 4\*1 + 2\*0 + 1\*0)

r-x – 5 (= 4\*1 + 2\*0 + 1\*1)

rwx – 7 (= 4\*1 + 2\*1 + 1\*1)

Как видно, атрибут *r* соответствует 4 в восьмиричной системе счисления (100 в двоичной), атрибут *w* – 2, атрибут *x* – 1. Складывая все веса, получаем числовое представление тройки прав. Но поскольку таких троек прав у нас три, в итоге получаем число, состоящее из трех цифр:

rwxr-xr-x – 755

rwx------ – 700

Нетрудно заметить, что в итоге у нас получается число в восьмеричной системе счисления – в ней как раз каждое знакоместо может содержать цифру от 0 до 7. Чтобы подчеркнуть восьмеричную природу числа, в программах на *PHP* его предваряют нулем:

rwxr-xr-x – 0755

rwx------ – 0700

#### Особенности атрибутов прав каталогов

Атрибут *x* для каталога имеет особое значение. Данный атрибут – разрешение просмотра атрибутов содержимого каталога.

Атрибут w разрешает создавать, переименовывать и удалять файлы внутри каталога. Заметьте, что не имеет значения, какие права установлены на файл, имя которого меняется (или удаляется). С удалением подкаталогов не все так очевидно – у вас должно быть право записи на удаляемый подкаталог.

Атрибут *r* разрешает вам получить список имен файлов и подкаталогов в каталоге, но не дает сведений о физическом расположении элементов.

Атрибут *x* как раз позволяет узнать физическое расположение элемента каталога по его имени. Например, вы можете перейти в каталог dir, имеющий атрибут x, при помощи команды UNIX cd dir (или вызова *chdir*("*dir*") в PHP). Вы также можете просмотреть содержимое файла *dir*/*somefile* (если, конечно, права доступа на файл это позволят), ведь информация о физическом положении файла somefile вам известна. Сложность механизма трактовки атрибутов каталогов, описанного выше, объясняет, почему на практике атрибуты r и x по отдельности встречаются довольно редко. Чаще всего их устанавливают (или снимают) одновременно.

### 6.3.2. Определение атрибутов файла

Функция *stat*() собирает вместе всю информацию, выдаваемую операционной системой об атрибутах указанного файла, и возвращает ее в виде массива.

array stat(string $filename)

Структура массива, возвращаемого функцией *stat*(), представлена в таблице 6.8.

**Таблица 6.8.** Структура массива, возвращаемого функцией *stat*().

|  |  |
| --- | --- |
| Индекс | Описание |
| 0 | Устройство |
| 1 | Номер узла *inode* |
| 2 | Атрибуты защиты файла |
| 3 | Число синонимов (жестких ссылок) файл |
| 4 | Идентификатор *UID* владельца |
| 5 | Идентификатор *GID* владельца |
| 6 | Тип устройства |
| 7 | Размер файла в байтах |
| **Индекс** | **Описание** |
| 8 | Время последнего доступа в секундах, прошедших с 01.01.1970 |
| 9 | Время последней модификации содержимого файла |
| 10 | Время последнего изменения атрибутов файла |
| 11 | Размер блока |
| 12 | Число занятых блоков |

В массив помещается информация, которая доступна в системах *UNIX*. Под *Windows* многие поля могут быть пусты (например, в файловой системе *FAT32* у файлов нет владельца, а значит, нет и идентификатора владельца файла и группы).

Если *$filename* задает не имя файла, а имя символической ссылки, то будет возвращена информация о том файле, на который указывает эта ссылка, а не о ссылке. Для получения информации о ссылке можно воспользоваться вызовом *lstat*()*,* имеющим точно такой же синтаксис, что и *stat*()*.*

### 6.3.3. Определение возможности доступа

Для определения возможности совершения какой-либо из операций над файлом, можно воспользоваться следующими функциями:

bool is\_readable(string $filename)

Возвращает *true*, если файл может быть открыт для чтения.

is\_writeable(string $filename) bool

Возвращает *true*, если в файл можно писать.

bool is\_executable(string $filename)

Возвращает *true*, если файл – исполняемый.

### 6.3.4. Функции для управления правами доступа

**Таблица 6.9.** Функции для управления и манипулирования правами доступа.

|  |  |
| --- | --- |
| Название функции | Описание |
| *fileowner* | Функция возвращает числовой идентификатор пользователя, владеющего указанным файлом (UID) |
| *chown* | Делает попытку сменить владельца файла $filename на указанного. Параметр $uid может быть числом (равным UID) или же строкой (содержащей имя пользователя в системе). В случае успеха возвращает true. |
| *chmod* | Функция предназначена для смены прав доступа к файл |
| *fileperms* | Функция возвращает числовое представление прав доступа к файлу |
| **Название функции** | **Описание** |
| *stat* | Функция собирает вместе всю информацию, выдаваемую операционной системой об атрибутах указанного файла, и возвращает ее в виде массива |
| *filesize* | Возвращает размер файла в байтах или false, если файла не существует |
| *filemtime* | Возвращает время последнего изменения содержимого файла или false в случае отсутствия файла |
| *fileatime* | Возвращает время последнего доступа (access) к файлу (например, на чтение). |
| *filectime* | Возвращает время последнего изменения атрибутов файла |
| *touch* | Устанавливает время модификации указанного файла |
| *filetype* | Возвращает строку, которая описывает тип файла |
| *is\_file* | Возвращает true, если $filename – обычный файл |
| *is\_dir* | Возвращает true, если $filename – каталог |
| *is\_link* | Возвращает true, если $filename – символическая ссылка |
| *is\_readable* | Возвращает true, если файл может быть открыт для чтения |
| *is\_writeable* | Возвращает true, если в файл можно писать |
| *is\_executable* | Возвращает true, если файл – исполняемый |
| *file\_exists* | Возвращает true, если файл с именем $filename существует на момент вызова |
| *readlink* | Функция возвращает имя основного файла, с которым связан его синоним |
| *symlink* | Эта функция создает символическую ссылку с именем на объект |

## 6.4. Взаимодействие с внешними ресурсами

*PHP* поставляется со множеством встроенных оберток для различных *URL*-протоколов для использования с функциями файловой системы, таких как *fopen*()*, copy*()*, file\_exists*()*, filesize*(), *file\_get\_contents*() и др. В дополнение к этим оберткам, можно регистрировать собственные обертки, используя функцию *stream\_wrapper\_register*()*.*

**Таблица 6.10.** Таблица поддерживаемых оберток.

|  |  |
| --- | --- |
| *file://* | Доступ к локальной файловой системе |
| *http://* | Доступ к *URL*-адресам по протоколу *HTTP(s)* |
| *ftp://* | Доступ к *URL*-адресам по протоколу *FTP(s)* |
| *php://* | Доступ к различным потокам ввода-вывода |
| *zlib://* | Сжатые потоки |
| *data://* | Схема *Data* (*RFC 2397)* |
| *glob://* | Нахождение путей, соответствующих шаблону |
| *phar://* | *PHP*-архив |
| *ssh2://* | *Secure Shell 2* |
| *rar://* | *RAR* |
| *ogg://* | Аудиопотоки |
| *expec://* | Потоки для взаимодействия с процессами |

Пример 6.12. Использование *HTTP* протокола.

<?php

//печатаем html-код страницы ya.ru на экран.

print file\_get\_contents("http://ya.ru");

При использовании оберток можно осуществлять достаточно сложное взаимодействие. Рассмотрим пример отправки *POST*-запроса на сервер и получения содержимого ответа.

Пример 6.13. Отправка *POST*-запроса.

<?php

$postdata = http\_build\_query(

array(

'var1' => 'value1',

'var2' => 'value2'

)

);

$opts = array('http' =>

array(

'method' => 'POST',

'header' => 'Content-Type: application/x-www-form-urlencoded',

'content' => $postdata

)

);

$context = stream\_context\_create($opts);

$result = file\_get\_contents('http://mydomain.com/submit.php', false, $context);

Рассмотрим основные функции для работы с обертками.

bool stream\_wrapper\_register ( string $protocol , string $classname [, int $flags = 0 ] )

Регистрирует обёртку *URL*, реализованную в виде *PHP*-класса. Позволяет вам реализовать ваши собственные обработчики протоколов и потоков для использования со всеми другими функциями файловой системы (такими как *fopen*()*, fread*() и т.д.). Возвращает *true* в случае успешного завершения или *false* в случае возникновения ошибки.

bool stream\_wrapper\_unregister ( string $protocol )

Позволяет вам отключить уже определённую обёртку потока. Как только обёртка будет отключена, вы можете перезаписать её пользовательской обёрткой.

bool stream\_wrapper\_restore ( string $protocol )

Восстанавливает встроенную обёртку, регистрация которой раньше была отменена при помощи *stream\_wrapper\_unregister*()*.*

# Глава 7. Регулярные выражения в формате PCRE

Регулярное выражение – это шаблон, с которым сравнивается строка слева направо. Шаблон пишется на специальном языке, позволяющем гибко описывать вид строки, который мы хотим найти.

Существуют различные виды регулярных выражений, однако в начтоящее время наиболее распространены регулярные выражения, совместимые с *Perl* (Perl Compatible Regular Expressions, *PCRE*). Как очевидно из названия, впервые они появились в языке *Perl* для удобства работы с конфигурационными файлами операционной системы Unix, однако оказались настолько удобными, что в последствии были включены во множество языков программирования.

Рассмотрим более подробно синтаксис и приемы с регулярными выражениями в формате *PCRE*.

## 7.1. Синтаксис языка PCRE

### 7.1.1. Ограничители

Язык *PCRE* требует, чтобы регулярное выражение было заключено между ограничителями – двумя слешами, также являющимися частью строки. Общий вид записи регулярного выражения – "/выражение/*M*", где *M – модификаторы (режимы).* Если символ "/" встречается в самом выражении, перед ним необходимо поставить обратный слеш "\", чтобы его экранировать.

Пример 7.1. Экранирование.

if (preg\_match('/path\\/to\\/file/i', "path/to/file"))

echo "Совпадение";

Если нам требуется писать регулярные выражения, содержащие большое количество слешей, то экранировать ихстановится неудобно, поэтому можно использовать альтренативные ограничители, такие как, например, "#". Тем не менее, если вы часто используете регулярные выражения лучше использовать один вид ограничителей, чтобы не возникало путаницы в том, что и когда вам нужно экранировать.

Пример 7.2. Альтернативные ограничители.

'/path\\/to\\/file/i'

'#path/to/file#i'

'{path/to/file}i'

### 7.1.2. Модификаторы

Модификаторы (или их еще называют режимы), нужны для того, чтобы настроить парсер регулярного выражения на определенный режим работы. Они располагаются после правого ограничителя регулярного выражения.

Доступные в *PHP* модификаторы представлены в таблице 7.1.

**Таблица 7.1.** Таблица поддерживаемых модификаторов.

|  |  |
| --- | --- |
| Режим | Описание |
| *i* | Игнорирование регистра. Например, при поиске по выражению /a+/i в строке "BAaB" найдется "Aa", а не "aa". |
| *x* | Пропуск пробелов и комментариев. Данный модификатор позволяет писать регулярные выражения в более изящном, читабельном виде. Допускается вставлять в выражение пробельные символы (в том числе перевод строки), а также однострочные комментарии, предваренные решеткой (#). |
| *m* | Многострочный режим работы. Входная строка разбивается на подстроки по символу конца строки. Поиск производится в каждой подстроке отдельно. |
| *s* | Однострочный поиск. Поиск производится во всей строке целиком. |
| *e* | Выполнение PHP-программы при замене. Модификатор работает только в функции замены *preg\_replace*() и заставляет *PHP* трактовать второй параметр ("на что заменять"), как код на *PHP*, результат работы которого подставляется вместо найденного участка. В настоящее время этот подход является устаревшим. |
| *u* | Данный модификатор переводит регулярные выражения в режим многобайтной кодировки *UTF*-8, его настоятельно рекомендуется включать, если работа идет с русским текстом. |

### 7.1.3. Отмена действия спецсимволов

Как вы увидите дальше, большое количество символов внутри регулярного выражения имеют специальное значение (например, ".", "+", "\*", "?", "[" и другие). Если нужно вставить в выражение один из управляющих символов, но чтобы он "не действовал", достаточно предварить его обратным слешем.

Для экранирования специальных символов существует функция *preg\_quote*().

string preg\_quote(string $str [,string $bound = NULL])

Она принимает на вход некоторую строку и экранирует в ней следующие символы:

. \ + \* ? [ ^ ] $ () { } = ! < > | : -

Дополнительно также экранируется символ, заданный в $bound (если указан). В списке, перечисленном выше, популярный ограничитель "/" не упоминается. Именно его и нужно писать в $bound в большинстве случаев.

### 7.1.4. Множества

В таблице 7.2 представлены основные способы задания множеств при написании регулярных выражений. Под описание, данное в множестве, подойдет ровно один символ при работе регулярного выражения.

**Таблица 7.2.** Способы описания множеств.

|  |  |
| --- | --- |
| Выражение | Описание |
| *a* | Одна буква "a". |
| *abc* | Подряд идущие "abc". |
| . | Один любой символ. |
| \*s* | Один пробельный символ. Существует большое количество подобных классов символов, например: \d – цифра, \w – буква или цифра. |
| \*S* | Один непробельный символ. Если после слеша большая буква – значит перед нами отрицание диапазона. Например, \D – не цифра. |
| [*abc*] | Одна из букв: "a", "b" или "c". |
| [*a*-*z*0-9] | Одна строчная английская буква или цифра. |
| [А-ЯЁа-яё] | Одна русская буква. Обратите внимание, что буква ё в таблице кодировки располагается отдельно от основного алфавита. |
| [*A*-*z*] | Одна английская буква. |
| [^0-9] | Не цифра. "^" внутри квадратных скобок имеет смысл отрицания. |
| [:alpha:] | Буква. Существует большое количество классов вида :класс:, например: ":digit:", "*:alnum*:", ":*blank*:" и другие. |

### 7.1.5. Мнимые символы

Мнимые символы – это участок строки между соседними символами, удовлетворяющий некоторым свойствам. Фактически мнимый символ – это некая позиция в строке.

**Таблица 7.3.** Примеры использования мнимых символов.

|  |  |
| --- | --- |
| Выражение | Описание |
| ^ | Соответствует началу строки (позиции перед первым символом). |
| $ | Соответствует концу строки (позиции за концом строки). |
| \*b* | Соответствует началу или концу слова. |
| \*B* | Любая позиция, кроме начала или конца слова. |
| ^*http*: | Соответствует строке, начинающейся с "http:". |
| \.*txt*$ | Подойдет для проверки того, что в строке хранится имя файла с раширением txt. |
| ^$ | Сопоставимо только с пустой строкой, потому что говорит: "сразу после начала строки идет ее конец". |

### 7.1.6. Квантификаторы повторений

Для указания того, сколько раз должен встретиться символ, описанный в множестве, используется один из квантификаторов повторений. Они представлены в таблице 7.3.

**Таблица 7.4.** Квантификаторы повторений.

|  |  |
| --- | --- |
| Выражение | Описание |
| ? | Один или ни одного. |
| + | Один и более. |
| \* | Ноль и более. |
| +? | "Нежадный" +. Захватывает меньший кусок строки при поиске совпадения. |
| \*? | "Нежадный" \*. Захватывает меньший кусок строки при поиске совпадения. |
| {5} | Ровно 5. |
| {2,} | 2 и более |
| {,8} | Не более 8. |
| {4,8} | От 4 до 8. |

Более подробно вопросы "жадности" квартификаторов будут раскрыты в разделе 7.4.

### 7.1.7. Комментарии в регулярных выражениях

При использовании модификатора *x*, неэкранированные пробелы, символы табуляции и пустой строки в шаблоне будут проигнорированы, если они не являются частью символьного класса. Также игнорируются все символы между неэкранированным символом '#' (если он не является частью символьного класса) и символом перевода строки (включая сами символы '\*n*' и '#').

Пример 7.3. Комментарии в режиме *x*.

preg\_match(

"/

(?(?=^[a-z]\*(?=:\/\/)) # Если есть протокол

(?:(?<protocol>^[a-z]\*)(?::\/\/)) # Протокол

...

)/xuis", $\_\_address, $parseAddressArr

);

Так же имеется возможность использовать служебную последовательность "(?#", которая обозначает начало комментария. Комментарий заканчивается закрывающейся круглой скобкой. Вложенные скобки не допускаются.

Пример 7.4. Комментарии в регулярном выражении.

$string = 'test';

echo preg\_match('/te(?# comments)st/', $string) . "\n";

## **7.2. Функции для работы с регулярными выражениями**

### 7.2.1. Функции поиска

Функция *preg\_match*() ищет в заданном тексте *subject* первое совпадение с шаблоном *pattern*. В случае, если дополнительный параметр *matches* указан, он будет заполнен результатами поиска.

mixed preg\_match (string pattern, string subject [, array &matches [, int flags [, int offset]]] )

Функция *preg\_match*() возвращает количество найденных соответствий. Это может быть 0 (совпадения не найдены) и 1, поскольку *preg\_match*() прекращает свою работу после первого найденного совпадения.

Если параметр *flags* установлен в *PREG\_OFFSET\_CAPTURE*, для каждой найденной подстроки будет указана ее позиция в исходной строке. Парметр *offset* отвечает за начальную позицию поиска.

Функция *preg\_match\_all*() ищет в строке *subject* все совпадения с шаблоном *pattern* и помещает результат в массив *matches* в порядке, определяемом комбинацией флагов *flags*. После нахождения первого соответствия последующие поиски будут осуществляться не с начала строки, а от конца последнего найденного вхождения.

int preg\_match\_all ( string pattern, string subject, array &matches [, int flags [, int offset]] )

Функция возвращает количество найденных вхождений шаблона (которое может быть и нулем) либо *FALSE*, если во время выполнения возникли какие-либо ошибки.

Дополнительный параметр *flags* может комбинировать следующие значения:

* *PREG\_PATTERN\_ORDER –* если этот флаг установлен, результат будет упорядочен следующим образом: элемент $*matches*[0] содержит массив полных вхождений шаблона, элемент $*matches*[1] содержит массив вхождений первой подмаски, и так далее;
* *PREG\_SET\_ORDER –* если этот флаг установлен, результат будет упорядочен следующим образом: элемент $*matches*[0] содержит первый набор вхождений, элемент $*matches*[1] содержит второй набор вхождений, и так далее;
* *PREG\_OFFSET\_CAPTURE –* если этот флаг указан, для каждой найденной подстроки будет указана ее позиция в исходной строке.

Пример 7.5. Поиск *E*-*mail* в тексте.

$text = "dfas me@yandex.ru dfasd gfdhjn him@google.com";

//результаты поиска попадут в массив $resultsArr

preg\_match\_all("/[a-z0-9\.\-]+@[a-z0-9\.\-]+/uis", $text, $resultsArr);

print\_r($resultsArr);

### 7.2.2. Функции замены

Функция *preg\_replace*() выполняет поиск в строке *subject* совпадений с шаблоном *pattern* и заменяет их на *replacement*.

mixed preg\_replace ( mixed pattern, mixed replacement, mixed subject [, int limit] )

В случае, если параметр *limit* указан, будет произведена замена *limit* вхождений шаблона; в случае, если *limit* опущен либо равняется -1, будут заменены все вхождения шаблона.

Пример 7.6. Удаление *E*-*mail* в тексте.

$text = "dfas me@yandex.ru dfasd gfdhjn him@google.com";

// Удаление – это замена на пустоту.

print preg\_replace("/[a-z0-9\.\-]+@[a-z0-9\.\-]+/uis", "", $text);

Функция *preg\_replace\_callback*() выполняет поиск по регулярному выражению и замену с использованием *callback*-функции.

mixed preg\_replace\_callback (mixed $pattern, callable $callback, mixed $subject [, int $limit = -1 [, int &$count ]])

Функция возвращает массив, если параметр *subject* является массивом, иначе возвращается строка. В случае ошибок возвращается *NULL*. Если найдены совпадения, будет возвращена результирующая строка, иначе *subject* вернется неизмененным.

*Callback*-функция должна вернуть строку с заменой. Достаточно часто *callback* функция, кроме как в вызове *preg\_replace\_callback*(), ни в чем больше не участвует. Исходя из этих соображений, можно использовать анонимные функции для создания *callback*-функции непосредственно в вызове *preg\_replace\_callback*().

Пример 7.7. Преобразование *E*-*mail* в нижний регистр.

$text = "dfas ME@yandex.ru dfasd gfdhjn HIM@google.com";

// Воспользуемся анонимной функцией.

print preg\_replace\_callback(

"/[a-z0-9\.\-]+@[a-z0-9\.\-]+/uis",

function ($matches) {

return mb\_strtolower($matches[0]);

},

$text

);

### 7.2.3. Функции фильтрации

Функция *preg\_grep*() возвращает массив, состоящий из элементов входящего массива *input*, которые соответствуют заданному шаблону *pattern*.

array preg\_grep ( string pattern, array input [, int flags] )

Параметр *flags* может принимать значение *PREG\_GREP\_INVERT.* В этом случае функция *preg\_grep*(), возвращает те элементы массива, которые не соответствуют заданному шаблону *pattern*.

Функция *preg\_filter*() производит поиск и замену по регулярному выражению.

mixed preg\_filter ( mixed $pattern , mixed $replacement , mixed $subject [, int $limit = -1 [, int &$count ]] )

Функция *preg\_filter*() идентична функции *preg\_replace*() за исключением того, что возвращает только те значения, в которых найдено совпадение.

Возвращает массив, если аргумент *subject* имеет тип *array* или строку в противном случае. Если совпадений не найдено или возникла ошибка, возвращается пустой массив, когда *subject* имеет тип *array* или *NULL* в противном случае.

### 7.2.4. Разбиение строки

Возвращает *preg\_split*() возвращает массив, состоящий из подстрок заданной строки *subject*, которая разбита по границам, соответствующим шаблону *pattern*.

array preg\_split ( string pattern, string subject [, int limit [, int flags]] )

В случае, если параметр *limit* указан, функция возвращает не более, чем *limit* подстрок. *flags* может быть произвольной комбинацией следующих флагов:

* *PREG\_SPLIT\_NO\_EMPTY*  *–* если этот флаг указан, функция *preg\_split*() вернет только непустые подстроки;
* *PREG\_SPLIT\_DELIM\_CAPTURE –* если этот флаг указан, выражение, заключенное в круглые скобки в разделяющем шаблоне, также извлекается из заданной строки и возвращается функцией;
* *PREG\_SPLIT\_OFFSET\_CAPTURE*  *–* если этот флаг указан, для каждой найденной подстроки, будет указана ее позиция в исходной строке

Пример 7.8. Разбиение текста на слова.

$text = "Пожалуйста, пришлите своё резюме или приходите в службу кадров пообщаться. Принесите с собой анкету";

// в качестве разделителя выберем пробелы и знаки препинания

$words = preg\_split("/[\s,.]+/uis", $text);

print\_r($words);

### 7.2.5. Таблица функций для работы с *PCRE*

**Таблица 7.5.** Функции для работы с *PCRE*.

|  |  |  |
| --- | --- | --- |
| № | Функция | Описание |
| 1 | *preg\_match\_all*() | Выполняет глобальный поиск шаблона |
| 2 | *preg\_match*() | Выполняет проверку на соответствие *PCRE* |
| 3 | *preg\_last\_error*() | Возвращает код ошибки выполнения последнего *PCRE* |
| 4 | *preg\_split*() | Разбивает строку по *PCRE* |
| 5 | *preg\_filter*() | Производит поиск и замену по *PCRE* |
| 6 | *preg\_grep*() | Возвращает массив вхождений, которые соответствуют *PCRE* |
| 7 | *preg\_replace*() | Выполняет поиск и замену по *PCRE* |
| 8 | *preg\_replace\_callback\_array*() | Выполняет поиск и замену по *PCRE* с использованием функций обратного вызова |
| 9 | *preg\_replace\_callback*() | Выполняет поиск по *PCRE* и замену с использованием *callback*-функции |

## 7.3. Карманы

### 7.3.1. Использование карманов

Часто при поиске нас интересует не только все выражение, но и его фрагменты. Например, при поиске E-mail адресов нам может быть интересно сразу выделить имена учетных записей пользователей (до знака @) и домены, на которых зарегистрированы почтовые ящики. Для решения этой задачи можно воспользоваться карманами.

Карман – это часть регулярного выражения, помещенная в круглые скобки. Использование карманов позволяет получить доступ к частям регулярного выражения в процессе выполнений различных операций.

Карманы нумеруются по номеру отрывающейся скобки и могут быть вложены друг в друга. Все выражение целиком нумеруется как нулевой карман. Рассмотрим пример.

Пример 7.9. Поиск *E*-*mail* с карманами.

$text = "dfas me@yandex.ru dfasd gfdhjn him@google.com";

//результаты поиска попадут в массив $resultsArr

preg\_match\_all("/([a-z0-9\.\-]+)@([a-z0-9\.\-]+)/uis", $text, $resultsArr);

print\_r($resultsArr);

//будет выведено

Array

(

[0] => Array

(

[0] => me@yandex.ru

[1] => him@google.com

)

[1] => Array

(

[0] => me

[1] => him

)

[2] => Array

(

[0] => yandex.ru

[1] => google.com

)

)

Как видно из примера, первый уровень результирующего массива отвечает за номер кармана, в котором уже располагаются соответствующие каждому совпадению части.

Так же карманы доступны и при замене по регулярному выражению. Допустим у нас стоит задача обрамить тегами <b> все слова, начинающиеся со знака доллар ("$").

Пример 7.10. Замена с использованием карманов.

$text = '$dfas dfasd $gfdhjn efs';

// Все выражение - нулевой карман!

print preg\_replace("/\\$[a-z]+/is", "<b>$0</b>", $text);

Вместо $0 для обращения к нулевому карману можно использовать "\\0".

### 7.3.2. Игнорирование карманов

Карманы нумеруются, начиная с индекса 1, причем карманом считается любое соответствие круглым скобкам. Иногда круглые скобки используются для группировки символов. Чтобы исключить такой карман из массива $*matches* или индекса, используемого для замены в функции *preg\_replace*(), применяется специальный синтаксис игнорирования карманов. Сразу после открывающейся круглой скобки указывается последовательность "?:".

Пример 7.11. Игнорирование карманов.

$str = "2015-12-15";

$re = '|^(?:\d{4})-(?:\d{2})-(\d{2})$|';

if (preg\_match($re, $str, $matches) {

echo htmlspecialchars("День: ".$matches[1]);

}

### **7.3.3. Именованные карманы**

Количество карманов может быть довольно велико, и их игнорирование не всегда возможно, особенно при отладке регулярного выражения. Поэтому для более удобного оперирования карманами введено их именование. Для этого после открывающейся круглой скобки указывается знак вопроса, после которого в угловых скобках или апострофах задается имя кармана**.**

Пример 7.12. Именованные карманы.

$str = "2015-12-15";

$re = "|^(?<year>\d{4})-(?<month> \d{2})-(?'day'\d{2})$|";

if (preg\_match($re, $str, $matches)) {

echo "День: " . $matches['day'] . "<br />";

echo "Месяц: " . $matches['month'] . "<br />";

echo "Год: " . $matches['year'] . "<br />";

}

## 7.4. Приемы работы с регулярными выражениями

### 7.4.1. Поиск с подстановкой

Очень часто возникает задача поиска, в которой необходимо использовать результаты того, что мы уже нашли. Например, нужно найти парные теги и важно, чтобы закрывающая часть тега была точно такая же, как открывающая. Эта задача решается с помощью поиска с подстановкой, когда мы используем в выражении ранее заполненные карманы.

Пример 7.13. Поиск парного тега.

$html = "<h1>Заголовок</h1>"

."<p style=\"color:black;\">абзац текста 1</p>"

."<p style=\"text-align:justify\">абзац текста 2</p>"

."<p class=\"find\_me\">абзац текста 3</p>";

//для обращения к содержимому первого кармана используем \\1

preg\_match\_all("/<\s\*([a-z0-9]+)[^>]\*>(.\*?)<\s\*\/\s\*\\1\s\*>/ui", $html, $resultArr);

print\_r($resultArr);

В данном примере с помощью подстановки "\\1" мы убеждаемся, что хвост тега совпадает с его началом.

### 7.4.2. Замена с подстановкой

При замене с помощью функции preg\_replace можно использовать части найденного выражения для формирования строки замены. Рассмотрим пример, когда все E-mail в тексте превращаются в активные ссылки, нажатие на которые приводит к открытию почтовой программы.

Пример 7.14. Замена *E*-*mail* на активные ссылки.

$text = "dfas me@yandex.ru dfasd gfdhjn him@google.com";

//Создадим вокруг E-mail тег <a href="maito:...">...</a>

print preg\_replace(

"/([a-z0-9\_\.\-]+)@[a-z0-9-]+\.[a-z]{2,4}/uis",

'<a href="mailto:$0">$1</a>',

$text

);

### 7.4.3. Незахватывающий поиск

Когда некоторое регулярное выражение или его часть внутри круглых скобок совпадает с подстрокой, оно "захватывает" эту подстроку, так что подвыражения, следующие далее, уже ее "не видят". Такое поведение не является обязательным: в *PCRE* существует целый ряд конструкций, позволяющих сравнивать подстроки без захвата.

#### Позитивный просмотр вперед

Оператор просмотра вперед – самая простая конструкция. Записывается так: (?=подвыражение)

На подвыражение в скобках не накладываются никакие ограничения: это может быть полноценное регулярное выражение. Когда в выражении встречается такая конструкция, текущая позиция считается допустимой, если с нее начинается подстрока, совпадающая с подвыражением в скобках. При этом "захвата" символов не происходит, и следующая конструкция будет работать с той же самой позицией в строке.

#### Негативный просмотр вперед

Существует также возможность негативного просмотра вперед – проверки, чтобы с текущей позиции не начиналось некоторое подвыражение. Записывается так: (?!подвыражение)

Например, если необходимо захватить все знаки пунктуации, кроме точки и запятой, то можно использовать регулярное выражение:

Пример 7.15. Пример негативного просмотра вперёд.

/(?![.,]) # дальше идет НЕ точка и НЕ запятая ([[:punct:]]+) # ...а какая-то другая пунктуация/x

Конструкцию (?!...) удобно использовать для быстрой проверки текущей позиции в регулярном выражении.

#### Позитивный просмотр назад

Просматривать строку без захвата символов можно не только вперед, но и назад. Для этого применяется следующий оператор: (?<=подвыражение).

Пример 7.16. Пример позитивного просмотра назад.

/(?<=<)# слева идет "<" – начало тега...

(\w+) # дальше - имя тега/x

#### Негативный просмотр назад

Негативный просмотр назад записывается он так: (?<!!подвыражение)

Вот пример из документации *PHP*. Выражение /(?<!*foo*)*bar*/ совпадает со строкой "*boobar*", но не совпадает со строкой "*foobar*".

### 7.4.4. "Жадность" квантификаторов.

Часто бывает так, что при использовании больших множеств поиск с помощью регулярных выражений выходит за границы искомого выражения. Это связано с "жадностью" кванторов. Чтобы более подробно разобрать этот вопрос, рассмотрим два примера.

Пример 7.17. "Жадные" квантификаторы.

$html = "<h1>some text. some text</h1>

<p class=\"myclass\">some text. some text</p>

<p>some text. some text</p>";

//Ищем содержимое тега <p>

preg\_match\_all("/<p[^>]\*>(.\*)<\/p>/uis", $html, $p\_arr);

print\_r($p\_arr);

//результат вывода на экран

Array

(

[0] => Array

(

[0] => <p class="myclass">some text. some text</p>

<p>some text. some text</p>

)

[1] => Array

(

[0] => some text. some text</p>

<p>some text. some text

)

)

Как видно из примера, вместо того, чтобы найти содержимое двух тегов p по отдельности, выражение ".\*" захватило содержимое соседнего тега в первое совпадение. Это связано с тем, что "\*" и "+" стараются захватить выражение по максимуму, но так, чтобы произошло совпадение. Нам же нужно минимально возможное совпадение, поэтому сделаем "\*" "нежадной", добавив к ней справа знак вопроса.

Пример 7.18. "Нежадные" квантификаторы.

$html = "<h1>some text. some text</h1>

<p class=\"myclass\">some text. some text</p>

<p>some text. some text</p>";

//Ищем содержимое тега <p>

preg\_match\_all("/<p[^>]\*>(.\*?)<\/p>/uis", $html, $p\_arr);

print\_r($p\_arr);

// Теперь мы нашли содержимое каждого тега в отдельности

Array

(

[0] => Array

(

[0] => <p class="myclass">some text. some text</p>

[1] => <p>some text. some text</p>

)

[1] => Array

(

[0] => some text. some text

[1] => some text. some text

)

)

# Глава 8. Работа с датой и временем в *PHP*

*PHP* обладает широкими возможностями для работы с датами и временем в различных форматах. Для того, чтобы функции работы с датой и временем работали правильно, необходимо правильно установить часовой пояс. Для этого в *php.ini* нужно задать директиву:

date.timezone = "Europe/Moscow";

Если по каким-то причинам доступ к *php.ini* затруднен, то можно вызвать в программе *PHP*-функцию:

date\_default\_timezone\_set("Europe/Moscow");

## 8.1. Функции времени в *PHP*

### 8.1.1. Формат времени *timestamp*

В формате *timestamp* время представляется как количество секунд, прошедшее с 01.01.1970 (начало этохи *UNIX*). То есть, по сути – это целое число, а значит даты в таком формате удобно сравнивать.

Для получения *timestamp* текущего времени используется функция *time*():

int time ( void )

Функция *microtime*() возвращает текущее время в формате *timestamp* с микросекундами.

mixed microtime ([ bool $get\_as\_float = FALSE ])

Если *get\_as\_float* = *true*, *microtime*() возвратит число с плавающей точкой (*float*) вместо строки (*string*).

Это дает нам возможность вычислять время работы сценария:

Пример 8.1. Время работы сценария *PHP*.

$start\_time = microtime(true);

// Тут расположен код сценария

$end\_time = microtime(true);

//Время работы - это разность между временем конца и начала работы

printf ("Время работы сценария: %0.4f секунд", $end\_time - $start\_time);

### 8.1.2. Строковое представление даты

Для построения строкового представления даты используется функция *date*():

string date( string $format [, int $timestamp *=* time() ] )

Функция возвращает строку, отформатированную в соответствии с указанным шаблоном *format*. Используется метка времени, задаваемая аргументом *timestamp*, или текущее системное время.

Обратите внимание, что результаты вывода для некоторых форматов зависят от текущей локали, которую можно задать функцией *setlocale*().

**Таблица 8.1.** Некоторые символы параметра $*format* функции *date*().

|  |  |  |
| --- | --- | --- |
| Символ | Описание | Возвращает значение |
| *d* | День месяца, 2 цифры с ведущим нулём | от 01 до 31 |
| *D* | Текстовое представление дня недели, 3 символа | от Mon до Sun |
| *J* | День месяца без ведущего нуля | от 1 до 31 |
| *l* | Полное наименование дня недели | от Sunday до Saturday |
| *N* | Порядковый номер дня недели в соответствии со стандартом ISO-8601 | от 1(понедельник) до 7 (воскресенье) |
| *z* | Порядковый номер дня в году (начиная с 0) | от 0 до 365 |
| *W* | Порядковый номер недели года в соответствии со стандартом ISO-8601; | Например: 42 (42-я неделя года) |
| *F* | Полное наименование месяца, например, January | от January до December |
| *m* | Порядковый номер месяца с ведущим нулём | от 01 до 12 |
| *M* | Сокращенное наименование месяца, 3 символа | от Jan до Dec |
| *n* | Порядковый номер месяца без ведущего нуля | от 1 до 12 |
| *t* | Количество дней в указанном месяце | от 28 до 31 |
| *L* | Признак високосного года | 1, если год високосный, иначе 0. |
| *Y* | Порядковый номер года, 4 цифры | Примеры: 1999, 2003 |
| *g* | Часы в 12-часовом формате без ведущего нуля | от 1 до 12 |
| *G* | Часы в 24-часовом формате без ведущего нуля | от 0 до 23 |
| *h* | Часы в 12-часовом формате с ведущим нулём | от 01 до 12 |
| *H* | Часы в 24-часовом формате с ведущим нулём | от 00 до 23 |
| *i* | Минуты с ведущим нулём | от 00 до 59 |
| *s* | Секунды с ведущим нулём | от 00 до 59 |

**Пример 8.2.** Текущее время**.**

<?php

print date("d.m.Y H:i:s");

### 8.1.3. Получение даты в виде массива

Функция ***getdate***() возвращает ассоциативный массив, содержащий информацию о дате, представленной меткой времени timestamp или текущим системным временем, если timestamp не был передан.

array **getdate** ([ int $timestamp = time()])

Все ключи возвращаемого функцией массива представлены в таблице 8.2.

**Таблица 8.2.** Ключи массива, возвращаемые *getdate*().

|  |  |  |
| --- | --- | --- |
| Символ | Описание | Возвращает значение |
| "*seconds*" | Числовое представление секунд | от 0 до 59 |
| "*minutes*" | Числовое представление минут | от 0 до 59 |
| "*hours*" | Числовое представление часов | от 0 до 23 |
| "*mday*" | Порядковый номер дня месяца | от 1 до 31 |
| "*wday*" | Порядковый номер дня недели | от 0 (воскресенье) до 6(суббота) |
| "*mon*" | Порядковый номер месяца | от 1 до 12 |
| "*year*" | Номер года, 4 цифры | Примеры: 1999, 2003 |
| "*yday*" | Порядковый номер дня в году | от 0 до 365 |
| "*weekday*" | Полное наименование дня недели | от *Sunday* до *Saturday* |
| "*month*" | Полное наименование месяца, например, *January* или *March* | от *January* до *December* |
| 0 | Количество секунд, прошедших с начала эпохи *Unix* | Зависит от платформы, в большинстве случаев от  -2147483648 до 2147483647. |

### 8.1.4. Построение timestamp

Таблица 7.3 Элементы ассоциативного массива *timestamp*

Функция mktime() возвращает метку времени *Unix*, соответствующую дате и времени, переданных в качестве аргументов. Аргументы могут быть опущены в порядке справа налево. В этом случае их значения по умолчанию равны соответствующим компонентам локальной даты/времени.

int mktime ([ int $hour = date("H") [, int $minute = date("i") [, int $second = date("s") [, int$month = date("n") [, int $day = date("j") [, int $year = date("Y") [, int $is\_dst = -1 ]]]]]]] )

Смысл парметров функции очевиден из описания функции, кроме параметра $*is\_dst*, которые можно установить в 1 или 0, для указания летнего или зимнего времени.

Для разбора строки, содержащей время, в *timestamp* можно воспользоваться функцией *strtotime*().

int strtotime ( string $time [, int $now = time()])

Первым параметром функции должна быть строка с датой, которая будет преобразована в timestamp относительно метки времени, переданной в *now*, или текущего времени, если аргумент *now* опущен.

Пример 8.3. Преобразование времени из строки.

<?php

print date("d.m.Y", strtotime("2019/01/12"));

//выведет 12.01.2019

### 8.1.5. Проверка даты

Проверить корректность даты по имеющимся году, месяцу и числу можно с помощью функции ***checkdate***():

bool **checkdate** ( int $month, int $day, int $year)

Проверяет корректность даты по переданным аргументам. Дата считается корректной, если все параметры принимают допустимые значения. Возвращает *TRUE*, если проверяемая дата является корректной, или *FALSE* в противном случае.

### 8.1.6. Дата и время по Гринвичу

До сих пор мы рассматривали так называемое локальноевремя *–* его показывают часы в том часовом поясе, где работает сервер. В реальной жизни время у пользователей, заходящих на web-страницы, отличается, поскольку они находятся в разных часовых поясах.

Для того чтобы не путаться в часовых поясах, придумали специальный формат време­ни – гринвичский (*Greenwich* *Mean* *Time*, *GMT*; еще одна аббревиатура, обозначаю­щая то же самое – *UTC*). Время по Гринвичу – это то время, которое в настоящий момент показывают часы в г. Гринвич, Англия. Там же проходит знаменитый "нулевой меридиан".

Для обозначения времени в других часовых поясах принята запись *gmt* +ччоо, где чч– разница времени в часах. Например, обозначение Москвы – *gmt* +0300. Это означает, что время в Москве на 3 часа отличается от времени нулевого меридиана (в большую сторону).

Удобство хранения времени в формате UTC заключается в том, что в случае переезда сервера, на котором развернуто приложение в другое место, не придется менять данные.

В *РНР* существует ряд функций, которые принимают в параметрах *локальное время* и возвращают различным образом оформленные даты, которые в текущий момент акту­альны на нулевом меридиане.

Это, например, функция *gmdate*()*,* предназначенная для получения строкового представления даты по *GMT*, или функция *gmmktime*() , создаю­щая *timestamp*-формат по указанной ей локальной дате. Функция *gm2local*() получает на вход *timestamp*-значение по Гринвичу и вычисляет, чему будет равен этот *timestamp* в указанном часовом поясе (по умолчанию – в текущем).

### 8.1.7. Функции даты и времени

Основные функции для работы с датой и временем представлены в таблице 8.3.

**Таблица 8.3.** Функции для работы с датой и временем.

|  |  |
| --- | --- |
| Функция | Описание |
| *checkdate* | Проверяет корректность даты по григорианскому календарю |
| *date\_default\_timezone\_get* | Возвращает временную зону, используемой по умолчанию всеми функциями даты/времени в скрипте |
| *date\_default\_timezone\_set* | Устанавливает временную зону по умолчанию для всех функций даты/времени в скрипте |
| *date\_parse\_from\_format* | Получение информации о заданной в определенном формате дате |
| *date\_parse* | Возвращает ассоциативный массив с подробной информацией о заданной дате |
| *date\_sun\_info* | Возвращает массив с информацией о закате/рассвете и начале/окончании сумерек |
| *date\_sunrise* | Возвращает время рассвета для заданных дня и местоположения |
| *date\_sunset* | Возвращает время захода солнца для заданных дня и местоположения |
| *date* | Форматирует вывод системной даты/времени |
| *getdate* | Возвращает информацию о дате/времени |
| *gettimeofday* | Возвращает текущее время |
| *gmdate* | Форматирует дату/время по Гринвичу |
| *gmmktime* | Возвращает метку времени *Unix* для времени по Гринвичу |
| *gmstrftime* | Форматирует дату/время по Гринвичу с учетом текущей локали |
| *idate* | Преобразует локальное время/дату в целое число |
| *localtime* | Возвращает локальное время |
| *microtime* | Возвращает текущую метку времени *Unix* с микросекундами |
| *mktime* | Возвращает метку времени *Unix* для заданной даты |
| *strftime* | Форматирует текущую дату/время с учетом текущих настроек локали |
| *strptime* | Разбирает строку даты/времени, сгенерированную функцией *strftime*() |
| *strtotime* | Преобразует текстовое представление даты на английском языке в метку времени *Unix* |
| *time* | Возвращает текущую метку времени *Unix* |
| *timezone\_name\_from\_abbr* | Возвращает временную зону в соответствии с аббревиатурой |
| *timezone\_version\_get* | Получение номера версии базы данных временных зон |

## 8.2. Календарные классы в PHP

В PHP реализовано несколько классов для работы с датой и временем. Наиболее используемыми являются *DateTime*, *DateInterval* и *DatePeriod*.

### 8.2.1. Класс DateTime

Основным классом для работы с датой и временем является класс *DateTime*.

**Основные методы класса**

*DateTime::add*– добавляет заданное количество дней, месяцев, лет, часов, минут и секунд к объекту *DateTime.*

*DateTime::createFromFormat* – разбирает строку с датой согласно указанному формату.

*DateTime::getLastErrors* – возвращает предупреждения и ошибки.

*DateTime::diff* – возвращает разницу между двумя объектами *DateTime.*

*DateTime::format* – возвращает дату, отформатированную согласно переданному формату.

*DateTime::getOffset* – возвращает смещение временной зоны.

*DateTime::getTimestamp* – возвращает временную метку *Unix.*

*DateTime::getTimezone* – возвращает временному зону относительно текущему значению *DateTime.*

*DateTime::modify* – изменение временной метки.

*DateTime::setDate* – устанавливает дату.

*DateTime::setISODate* – устанавливает дату в формате *ISO.*

*DateTime::setTime* – устанавливает время.

*DateTime::setTimestamp* – устанавливает дату и время на основе метки времени *Unix.*

*DateTime::setTimezone* – устанавливает временную зону для объекта класса *DateTime.*

*DateTime::sub* – вычитает заданное количество дней, месяцев, лет, часов, минут и секунд из времени объекта *DateTime.*

Рассмотрим пример использования класса DateTime.

Пример 8.4. Использование класса *DateTime*.

<?php

$date = new DateTime('2000-01-01');

//добавляем день

$date->modify('+1 day');

//устанавливаем дату и время

$date->setDate(2001, 2, 3);

$date->setTime(14, 55, 24);

$date->setTimezone(new DateTimeZone('Pacific/Chatham'));

//вычитаем 10 дней

$date->sub(new DateInterval('P10D'));

### 8.2.2. Класс DateInterval

Отличительной особенностью объектов класса *DateTime* является возможность вычи­тать их друг из друга при помощи метода *diff* ()*.* Кроме того, можно добавлять и вычи­тать из объекта *DateTime* временные интервалы, соответственно при помощи методов *add*() и *sub*()*.* Для обеспечения этих операций в наборе календарных классов РНР пре­дусмотрен класс *Datelnterval*.

**Основные методы класса:**

*DateInterval::\_\_construct* – создает новый объект *DateInterval.*

*DateInterval::createFromDateString* – создает объект класса *DateInterval* из даты в относительном формате.

*DateInterval::format* – форматирует интервал.

**Таблица 8.4.** Форматы описания периодов.

|  |  |
| --- | --- |
| Период | Описание |
| *Y* | года |
| *M* | месяцы |
| *D* | дни |
| *W* | недели. Преобразуется в дни, поэтому не может быть использован совместно с *D*. |
| *H* | часы |
| *M* | минуты |
| *S* | секунды |

Формат описания интервала начинается с буквы *P*. Каждый период интервала представлен целым числом, за которым следует указатель его типа. Возможные типы приведены в таблице. Если интервал содержит периоды, обозначающие время, то их описанию должна предшествовать буква *T*.

**Пример 8.5.** Использование класса *DateInterval*.

//2 года, 4 дня, 6 часов, 8 минут

$interval = new DateInterval('P2Y4DT6H8M');

echo $interval->format('%d days');

//выведет 4 дня

### 8.2.3. Класс DatePeriod

Объект класса *DatePeriod* позволяет создать итератор для обхода последовательности дат (в виде *DateTime*-объектов), следующих друг за другом через определенный интер­вал времени.

Конструктор класса *DatePeriod* принимает три параметра:

1. Объект *DateTime*, который служит точкой начала периода.
2. Объект *DateInterval*, служащий шагом, с которым генерируются даты.
3. Целое число, представляющее количество итераций.

**Основные методы класса:**

*DatePeriod::\_\_construct* – Создает новый объект *DatePeriod*

*DatePeriod::getDateInterval* – Возвращает интервал

*DatePeriod::getEndDate* – Возвращает конечную дату периода

*DatePeriod::getStartDate* – Возвращает начальную дату периода

Пример 8.6. Использование класса *DatePeriod*.

<?php

//создаем начальный и конечный день

$begin = new DateTime('2012-08-01');

$end = new DateTime('2012-08-31');

$end = $end->modify('+1 day');

//задаем интервал в 1 день

$interval = new DateInterval('P1D');

//делаем диапазон дат

$daterange = new DatePeriod($begin, $interval ,$end);

//выводим все даты диапазона

foreach($daterange as $date){

echo $date->format("d.m.Y") . "<br/>";

}

# Глава 9. Расширенные возможности языка PHP

## 9.1. Взаимодействие с внешними программами из *PHP*

### 9.1.1. Запуск утилит

**Функция *system*()** выполняет указанную команду *command* и выводит её результат.

string **system** (string $command [, int &$return\_var ] )

Вызов функции ***system*()** также пытается автоматически очистить буфер вывода веб-сервера после каждой строки вывода, если *PHP* работает как модуль сервера. Если вам нужно выполнить команду и получить все данные из команды в окно браузера, используйте функцию *passthru*()*.*

Так же для выполнения команд может быть использована функция *exec*():

string **exec** ( string $command [, array &$output [, int &$return\_var ]] )

Функция возвращает последнюю строку из результата вывода команды. Если параметр *output* указан, то этот массив будет заполнен строками вывода программы. Обратите внимание, что если массив уже содержит какие-либо элементы, то ***exec*()** добавит новые элементы в конец массива.

Пример 9.1. Использование оператора обратные апострофы.

//узнаем кодировку консоли windows (866)

print system("chcp");

//берем список папок

exec("dir", $dirs);

//конвертируем bp cp866 в Utf-8

foreach($dirs as $dir)

print mb\_convert\_encoding ($dir, "utf-8", "cp866")."\n";

//результат вывода

…: 866

…: 866 Том в устройстве C не имеет метки.

Серийный номер тома: 2674-5B7B

Содержимое папки C:\dev\test

19.10.2019 13:50 <DIR> .

19.10.2019 13:50 <DIR> ..

19.08.2019 19:16 3 007 632 my.mp3

23.08.2019 19:07 117 265 all\_outer.m3u

19.11.2019 12:15 14 394 index.php

14.08.2019 18:22 <DIR> nbproject

15.11.2019 12:21 <DIR> songs

3 файлов 3 139 291 байт

2 папок 24 805 134 336 байт свободно

Функция ***passthru*()** похожа на функцию *exec*() в том, что она выполняет команду command. Эта функция должна быть использована вместо функции *exec*() или *system*()*,* когда вывод команды *Unix* представляет собой двоичные данные, которые необходимо передать непосредственно в браузер.

void passthru ( string $command [, int &$return\_var ] )

Стандартное использование этой функции – выполнение утилит, которые выводят непосредственно поток изображения. Установив *Content-type* в *image/gif*, вы можете создать *PHP*-скрипты, которые выводят изображения напрямую в браузер.

#### Оператор обратные апострофы

В *РНР* существует специальный оператор – обратные апострофы – для запуска внешних программ и получения результата их выполнения. То есть оператор `` возвращает данные, отправленные запущенной программой в стандартный выходной поток:

Пример 9.2. Использование оператора обратные апострофы.

echo `dir`;

//Результат вывода

Microsoft Windows [Version 10.0.18362.476]

Данный пример выводит в браузер результат команды *ver*, которая доступна в *Windows* и предназначена для распечатки версии операционной системы.

### 9.1.2. Экранирование командной строки

Команда ***escapeshellcmd*()** экранирует символы, которые могут быть использованы для обмана командной оболочки при выполнении команд.

string **escapeshellcmd** ( string $command )

Эту функцию целесообразно использовать, чтобы убедиться, что любые данные, вводимые пользователем, будут экранированы перед передачей их функциям*exec*() или *system*() или оператору "обратныеапострофы".

Следующие символы будут экранированы при помощи обратного слеша: &#;`|\*?~<>^*()[]{}*$\, \*x0A* и \*xFF*. Символы ' и " экранируются только в том случае, если они встречаются не попарно. В *Windows* все эти символы, а так же "!" и "%" заменяются пробелом.

Функция ***escapeshellarg*()** добавляет одинарные кавычки вокруг строки и кавычек и экранирует любые существующие одинарные кавычки, позволяя вам передать строку непосредственно в функцию оболочки и рассматривать её как один безопасный аргумент.

string **escapeshellarg** ( string $arg )

Экранирование аргумента необходимо в таких функциях оболочки как *exec*()*, system*() и оператор "обратныеапострофы".

### 9.1.3. Каналы

Мы уже привыкли, что можем открыть некоторый файл для чтения при помощи функ­ции *fopen*()*,* а затем читать или писать в него данные. Хотелось бы иметь механизм, который позволял бы делать что-то подобное с запущенной программой.

Можно, конечно, заранее сохранить данные для запроса в отдельном временном файле, затем запустить программу, указав ей этот файл в качестве стандартного входного по­тока (оператор < оболочки):

Пример 9.3. Передача параметров через файл.

$tmp = "parameters.txt";

file\_put\_contents($tmp, "Hello");

system("commandToExecute < $tmp");

unlink($tmp);

Такой подход неудобен, кроме того, создание временного файла требует дополнительных ресурсов. В такой ситуации и удобно ис­пользовать межпроцессные каналы. Открытие канала происходит с помощью функции *popen*().

resource popen(string $cmd, string $mode)

Функция *popen*() возвращает файловый указатель, идентичный возвращаемому функцией *fopen*(), за исключением того, что он односторонний (может быть использован только для чтения или записи) и должен быть закрыт при помощи *pclose*().Если в качестве режима указано '*r*', файловый указатель аналогичен потоку вывода (*STDOUT*) команды, если же указано '*w*', то файловый указатель аналогичен потоку ввода (*STDIN*) команды.

Пример 9.4. Чтение вывода внешней программы через канал.

$handle = popen("/bin/ls", "r");

$read = fread($handle, 2096);

echo $read;

pclose($handle);

Обратите внимание, при помощи *рореn*() канал нельзя открыть в режи­ме одновременного чтения и записи. Тем не менее в РНР существует функция *proc\_open*(), которая умеет запускать процессы и позволяет при этом работать как с их входным, так и с выходным потоками.

resource proc\_open ( string $cmd , array $descriptorspec, array &$pipes [, string $cwd [, array $env]] )

Параметры функции *proc\_open*() следующие:

**Таблица 9.1.** Параметры функции *proc\_open*().

|  |  |
| --- | --- |
| Период | Описание |
| *cmd* | Команда для выполнения. |
| *descriptorspec* | Массив, ключи которого представляют собой номер дескриптора, а значение описывает, как PHP должен передать этот дескриптор дочернему процессу. 0 – stdin, 1 – stdout и 2 – stderr. |
| *pipes* | Будет задан массивом указателей на файлы, соответствующие созданным каналам передачи данных PHP. |
| *cwd* | Рабочая директория команды. Это должен быть абсолютный путь к директории или NULL, если требуется использовать директорию по умолчанию (рабочая директория текущего процесса PHP). |
| *env* | Массив переменных окружения для запускаемой команды или NULL, если требуется использовать то же окружение, что и у текущего PHP-процесса. |
| *other\_options* | Только для Windows. |

*proc\_open*() возвращает ресурс, представляющий процесс. Этот ресурс необходимо освобождать функцией *proc\_close*() по завершении работы с ним.

Пример 9.5. Взаимная блокировка с *proc\_open*().

<?php

$spec = [

0 => ["pipe", "г"], // stdin

1 => ["pipe", "w"], // stdout

2 => ["file", "/tmp/error-output.txt", "a"] // stderr

];

// Запускаем процесс

$proc = proc\_open("cat", $spec, $pipes);

// Дальше можно писать в $pipes[0] и читать из $pipes[1]

for ($i = 0; $i < 100; $i++)

fwrite($pipes[0], "Hello World #$i!\n");

fclose($pipes[0]);

while (!feof($pipes[1]))

echo fgets($pipes[1], 1024);

fclose ($pipes[1]);

// Закрываем дескриптор

proc\_close($proc);

Используйте функцию *proc\_open*() очень осторожно: при неаккуратном применении возможна взаимная блокировка (*deadlock*) скрипта и вызываемой им программы. Представьте, что запущенная утилита принимает данные из своего входного потока и тут же перенаправляет их в выходной поток.

Именно так поступает команда *UNIX* *cat*, которая задействована в примере выше. Когда мы записываем данные в $*pipes*[0], утилита немедленно перенаправляет их в $*pipes*[*i*], где они накапливаются в буфере.

Но размер буфера не безграничен – обычно всего 10 Кбайт. Как только он заполнится, утилита *cat* войдет в состояние сна (в функции записи): она будет ожидать, пока кто-нибудь не считает данные из буфера, таким образом, освободив немного места.

Утилита ждет, пока сценарий считает данные из ее выходного потока, а скрипт – пока утилита будет готова принять информацию, которую он ей передает. Фактически две программы ждут друг друга, и это может продолжаться до бесконечности.

## 9.2. Управление интерпретатором PHP

### 9.2.1. Информационные функции

Функция ***phpinfo***() выводит большое количество информации о текущем состоянии *PHP*.

bool **phpinfo** ([ int $what = INFO\_ALL ] )

Выводится информация о настройках компиляции *PHP*, о расширениях, о версии, о сервере и среде выполнения, окружении *PHP*, версии ОС, о путях, об основных и локальных значениях настроек конфигурации и т.д.

Так как каждая система имеет свои особенности, ***phpinfo*()** используется в основном для проверки настроек конфигурации и для просмотра доступных предопределенных констант. Для уточнения вывода, можно задать параметр *what*.

**Таблица 9.2.** Настройки параметра *what* функции *phpinfo*().

|  |  |  |
| --- | --- | --- |
| Имя (константа) | Значение | Описание |
| *INFO\_GENERAL* | 1 | Строка конфигурации, расположение php.ini, дата сборки, Web-сервер, Система и др. |
| *INFO\_CREDITS* | 2 | Разработчики PHP.  См. также *phpcredits*(). |
| *INFO\_CONFIGURATION* | 4 | Текущие значение основных и локальных PHP директив.  См. также *ini\_get*(). |
| *INFO\_MODULES* | 8 | Загруженные модули и их настройки.  См. также *get\_loaded\_extensions*(). |
| *INFO\_ENVIRONMENT* | 16 | Информация о переменных окружения, которая также доступна в $*\_ENV*. |
| *INFO\_VARIABLES* | 32 | Выводит все предопределенные переменные из *EGPCS* (Environment, *GET*, *POST*, *Cookie*, *Server*). |
| *INFO\_LICENSE* | 64 | Информация о лицензии *PHP*. |
| *INFO\_ALL* | -1 | Выводит все приведенное выше. |

Для определения версии PHP можновоспользоваться функцией ***phpversion***():

string **phpversion** ([ string $extension ] )

Для получения времени последней модификации запущенного скрипта, используйте функцию ***getlastmod***():

int **getlastmod** ( void )

Если требуется получить время модификации произвольного файла, используйте функцию *filemtime*()*.*

Чтобы получить информацию о потребляемой памяти, используйте функцию *memory\_get\_usage*().

int memory\_get\_usage ([ bool $real\_usage = FALSE ] )

Передача *TRUE* позволяет узнать реальное количество памяти, выделенной *PHP* скрипту системой, включая неиспользуемые страницы. Если аргумент не задан или равен *FALSE*, будет возвращено только количество используемой памяти.

### 9.2.2. Принудительное завершение программы

Функция *exit*() прекращает выполнение скрипта.

void **exit** (mixed $status )

При вызове функции, будут вызваны финализаторы и деструкторы объектов. *Exit* – это конструкция языка, и она может быть вызвана без круглых скобок, если не передается параметр *status*.

Если *status* задан в виде строки, то эта функция выведет содержимое *status* перед выходом. Если *status* задан в виде целого числа (0-254), то это значение не будет выведено. Статус выхода 0 используется при успешном завершения программы.

Функция *die*() делает почти то же самое, что и *exit*(), только перед завершением работы выводит строку, заданную в параметре *message.* Чаще всего ее применяют, если нужно напечатать сообщение об ошибке и аварийно завершить программу.

void die(string $message)

Полезным примером использования *die*() может служить такой код:

Пример 9.6. Прерывание программы с помощью *die*().

$file = @fopen("file.txt", 'r')

or die("не могу открыть файл file.txt!");

В программах PHP можно задать ограничение на время выполнения скрипта. Это делается с помощью функции set\_time\_limit().

bool set\_time\_limit (int $seconds)

Функция задает время в секундах, в течение которого скрипт должен завершить работу. Если скрипт не успевает, вызывается фатальная ошибка. По умолчанию дается 30 секунд, либо время, записанное в настройке *max\_execution\_time* в *php.ini*.

При вызове ***set\_time\_limit*()** перезапускает счетчик с нуля. Другими словами, если тайм-аут изначально был 10 секунд, и через 5 секунд после запуска скрипта будет вызвана функция *set\_time\_limit*(20), то скрипт будет работать максимум 25 секунд.

### 9.2.3. Финализаторы

Разработчики *РНР* предусмотрели возможность указать в программе функцию-финализатор, которая будет автоматически вызвана, как только работа сценария завершится. В такой функции можно, например, обновить файл журнала работы программы.

Для создания финализатора нужно написать функцию и зарегистрировать ее как финализатор с помощью функции *register\_shutdown\_function*()*.*

void **register\_shutdown\_function** ( callable $callback [, mixed $parameter [, mixed $... ]] )

Регистрирует функцию *callback*, которая выполнится после завершения работы скрипта или при вызове функции *exit*().

Возможна регистрация нескольких подобных функций с помощью ***register\_shutdown\_function()****,* при этом функции будут выполняться в том порядке, в каком они были зарегистрированы. Если вы вызовете *exit*() в одной из зарегистрированных завершающих функций, процесс будет полностью остановлен и последующие завершающие функции не будут вызваны.

### 9.2.4. Функции задержки выполнения программы

Обычно в PHP для создания задержек используют две функции – *usleep*() и *sleep*(). Функции откладывают исполнение программы на указанное количество микросекунд и секунд соответственно.

void **usleep** ( int $micro\_seconds )

int **sleep** ( int $seconds )

## 9.3. Генерация кода во время выполнения программы

В процессе исполнения *PHP*-кода могут быть сформированы строки сами по себе являющиеся *PHP*-кодом, которые можно отправить на выполнение. Такая возможность довольно широко используется, например, в различных программах-упаковщиках.

Для выполнения кода, содержащегося в строке используется функция *eval*().

mixed **eval** ( string $code )

Передаваемый код не должен быть обернут открывающимся и закрывающимся тегами *PHP*. Код должен быть верным, т.е. операторы должны быть разделены точкой с запятой и т.п.

Исполняемый код из строки будет выполняться в области видимости кода, вызвавшего ***eval*()***.* Таким образом, любые переменные, определенные или измененные в вызове ***eval*()***,* будут доступны после его выполнения в теле программы.

Функция ***eval*()** возвращает значение, переданное *return* или *NULL*.

Иногда для генерации кода требуется сгенерировать полноценной объявление некоторой переменной. Для этого используется функция *var\_export*().

mixed var\_export (mixed $expression[, bool $return = FALSE])

Если параметр return равен *TRUE*, ***var\_export*()** вернет представление переменной вместо его вывода.

Пример 9.7. Экспортирование переменной.

<?php

$numbers = [];

for ($count = 0; $count < 5; $count++) {

$numbers[] = rand(0, 9);

}

//экспортируем переменную

var\_export($numbers);

//результат вывода:

array (

0 => 9,

1 => 6,

2 => 5,

3 => 4,

4 => 4,

)

# Глава 10. Объектно-ориентированное программирование в PHP

## 10.1. Классы

Класс ­­– это базовое понятие в объектно-ориентированном программировании (ООП). Классы образуют синтаксическую базу ООП. Их можно рассматривать как своего рода "контейнеры" для логически связанных данных и функций (обычно называемых методами).

Экземпляр класса – это объект. Объект – это совокупность данных (свойств) и функций (методов) для их обработки. Свойства и методы называются членами класса.

### 10.1.1. Создание класса

Новый класс (тип данных) в программе описывается при помощи ключевого слова *class*. Внутри класса могут располагаться его свойства (переменные класса) и методы (функции-члены)

Данные описываются с помощью служебного слова *var*. Метод описывается так же, как и обыкновенная пользовательская функция. Методу также можно передавать параметры.

Пример 10.1. Объявление класса.

// Создаем новый класс Coor:

class Coor

{

// данные (свойства):

var $name;

var $addr;

// методы:

function name()

{

echo "<h3>John</h3>";

}

}

После того, как класс описан, можно создать объект. Для создания объекта необходимо воспользоваться ключевым словом new.

Пример 10.2. Создание объекта.

// Создаем объект класса Coor:

$object = new Coor;

Для получения доступа к членам класса существует оператор "–>". Чтобы получить доступ к членам класса внутри класса, необходимо использовать указатель $this, который всегда относится к текущему объекту. Указатель $this можно также использовать для доступа к методам, а не только для доступа к данным:

Пример 10.3. Обращение к свойствам и методам класса.

// Создаем новый класс Coor:

class Coor

{

// данные (свойства):

var $name;

// методы:

function getName()

{

echo $this->name;

}

function setName($\_\_name) {

$this->name = $\_\_name;

}

}

// Создаем объект класса Coor:

$object = new Coor;

// Изменяем имя

$object->setName("Nick");

// Выводим имя

$object->getName();

// Сценарий выводит "Nick"

При вызове метода класса первым делом создается локальная переменная $this, которой присваивается то же значение, что было у $object. То есть, в $this теперь хранится ссылка на объект, для которого вызывается метод. Вызов метода некоторого объекта автоматически предоставляет ему доступ к свойствам этого объекта посредством специальной переменной $this.

### 10.1.2. Инициализация и разрушение объектов

*Конструктор класса* вызывается всякий раз, когда используется оператор new для объекта. Для создания конструктора нужно объявить метод *\_\_construct*(). В *PHP* у класса может быть только один конструктор.

Как и для обычных функций и методов, для конструкторов можно задавать параметры по умолчанию.

Пример 10.4. Конструктор с параметрами по умолчанию.

function \_\_construct($re = 0, $im = 0) {

$this->re = $re;

$this->im = $im;

}

Деструктор – специальный метод объекта, который вызывается при уничтожении этого объекта (например, после завершения программы). Деструкторы обычно выполняют служебную работу – закрывают файлы, записывают протоколы работы, разрывают соединения, "форматируют жесткий диск" – в общем, освобождают ресурсы.

*Деструктор* – это специальный метод класса с именем *\_\_destruct*(), который будет гарантированно вызван при потере последней ссылки на объект в программе. Так как деструктор запускается самим *PHP*, он не должен принимать никаких параметров.

На объект могут ссылаться сразу несколько переменных. Объекты, на которые в программе не осталось ссылок, *PHP* немедленно удаляет из памяти (предварительно вызвав деструкторы). Для определения того, что ссылается ли что-то на объект или нет существует счетчик ссылок, применяемый в *PHP*.

Любой объект, который создается, содержит в себе скрытое поле, хранящее так называемый счетчик ссылок. Каждый раз, когда в программе появляется новая ссылка на объект, этот счетчик увеличивается на 1. Соответственно, при удалении ссылки счетчик уменьшается на 1. При обнулении счетчика на объект больше никто не ссылается, а потому его можно спокойно удалить из памяти, что *PHP* и делает. Таким образом, объект удаляется после некоторой операции присваивания (или вызова *unset*()), приводящей к потере последней ссылки на него.

### 10.1.3. Доступ к членам класса

#### Модификаторы доступа

В *PHP* существуют три модификатора ограничения доступа: *public*, *protected* и *private*. Их можно указывать перед описанием метода или свойства класса.

Члены класса, помеченные ключевым словом *public* ("публичный"), доступны для использования извне.

Пример 10.5. Публичные свойства класса.

class Hotel

{

public $exit;

public function escape()

{

echo "Пойдем вокруг {$this->exit}!";

}

}

$house = new Hotel();

$house->exit = "газона"; // допустимо

$house->escape(); // допустимо

С использованием ключевого слова *private* ("приватный") возможно сделать члены касса "невидимыми" из вызывающей программы, будто бы их и нет. В то же время, методы класса могут обращаться к ним без всякого ограничения.

Пример 10.6. Создание класса с модификатором доступа *private.*

class Hotel {

private $exit = "дома";

public function escape()

{

echo "Пойдем вокруг {$this->exit}!";

}

private function lock()

{

$this->exit = ", но везде закрыто!";

}

}

$house = new Hotel();

$house->exit = "газона"; // Ошибка!

$house->escape(); // Выполнится

$house->lock(); // Ошибка!

Модификатор *protected* ("защищенный") с точки зрения вызывающей программы выглядит точно так же, как и *private*: он запрещает доступ к членам объекта извне. Однако по сравнению с *private* он более "мягкий", поскольку позволяет обращаться к членам не только из "своих" методов, но также и из методов производных классов. Защищенными обычно делают лишь методы, но не свойства классов. Это позволяет создавать функции, которые, с одной стороны, не должны быть видны в основной программе, а с другой, могут использоваться в классах-потомках.

#### Неявное объявление свойств

Одна из особенностей *PHP* это то, что операторы:

$this->property = 101;

$obj->property = 303;

допустимы и не вызывают ошибку даже в случае, если свойство $*property* в классе не объявлено.

При этом в объекте $*obj* создастся свойство с именем $*property* и значением 303.

#### Статические свойства и методы

Классы выступают в качестве шаблонов для объектов. Для того чтобы обратиться к внутреннему содержимому объекта, используется ключевое слово $*this*. Для обращения к внутреннему содержимому класса используется ключевое слово *self*. Как видно, ключевое слово $*this* снабжено символом доллара, чтобы подчеркнуть связь с переменными. В то время как ключевое слово *self* обходится без символа доллара – это указание на то, что обращение идет не к переменной. Для того чтобы воспользоваться *self*, потребуется объявить статическую переменную или метод класса.

Особенностью статических членов и методов является тот факт, что они определяются не на уровне объекта, а на уровне класса. Статическое свойство недоступно через обращение $*this*–>*property* или $*obj*–>*property*. Вместо этого используется оператор "::" и либо имя класса (ИмяКласса::$*property*), либо ключевое слово *self* (*self*::$*property*). Статический метод во время своего запуска не получает ссылку $*this*, поэтому он может работать только со статическими членами (свойствами и другими методами) своего класса.

В *PHP* есть возможность определить метод как статический.. Статические свойства и методы принадлежат классам, а не объектам.

Пример 10.7. Статические свойства и методы.

class Math

{

private static $count = 0;

public function \_\_construct()

{

self::$count++;

}

public static function calcSin($x)

{

return sin($x);

}

public static function calcSQRT($x)

{

return sqrt($x);

}

public static function getCount()

{

return self::$count;

}

}

echo Math::calcSin(1)."<br/>";

echo Math::calcSQRT(9)."<br/>";

$math\_1 = new Math();

$math\_2 = new Math();

$math\_3 = new Math();

$math\_4 = new Math();

echo "Создано объектов: " . Math::getCount();

//Результат работы сценария:

0.8414709848079

3

Создано объектов: 4

Этот класс предоставляет инструменты для работы с математическими функциями без необходимости создания объекта. В классе есть конструктор, который позволяети считать количество созданных объектов в статическом свойстве $*count*. Обратите внимание, значение этого свойства явлется общим для всех объектов типа *Math*. Эта особенность статических свойств позволяет объектам одного типа общаться между собой.

Для объявления метода или свойства статическим используется слово *static*, а для доступа к статическому свойству используется слово *self* с двойным двоеточием "::".

#### Константы класса

Наряду с членами классы могут содержать константы, которые определяются при помощи ключевого слова *const*.

Пример 10.8. Объявление константы класса.

class Cls

{

const NAME = "cls";

public function method()

{

// echo $this->NAME; - так нельзя!

echo self::NAME;

}

}

Точно так же, как и в случае со статическими членами классов, к константам нельзя обращаться при помощи оператора "–>"; для обращения используется оператор разрешения области видимости "::", который предваряется либо именем класса, либо ключевым словом *self*. Существование констант может быть проверено при помощи функции *defined*(), которая возвращает *true*, если константа существует, и *false* в противном случае.

#### Перехват обращения к методам класса

*PHP* позволяет перехватывать обращения к несуществующим членам объекта. Для этого в класс необходимо добавить специальные методы, имена которых начинаются с двойного подчерка.

*\_\_get*() – перехват чтения несуществующего свойства класса.

*\_\_set*() – перехват присваивания в несуществующее свойство класса.

*\_\_call*() – перехват вызова несуществующей функции.

Перехватчики *\_\_get*(), *\_\_set*() и *\_\_call()* не всегда определены. По умолчанию они генерируют сообщение об ошибке: "попытка обращения к несуществующему члену класса". При добавлении своих перехватчиков такое поведение изменяется.

Пример 10.9. Класс с перехватом обращений.

class Point

{

private $x;

private $y;

public function \_\_construct($x, $y)

{

$this->x = $x;

$this->y = $y;

}

public function \_\_get($name)

{

echo "Произошло обращение к свойству $name<br />";

return $this->$name;

}

}

$p = new Point(8, 16);

echo "x: $p->x<br />";

echo "y: $p->y<br />";

echo "Несущесвующие поле: $p->nonexistentProperty";

//Результат работы сценария:

Произошло обращение к свойству x

x: 8

Произошло обращение к свойству y

y: 16

Произошло обращение к свойству nonexistentProperty

### 10.1.4. Клонирование объектов

В *PHP* объекты представляют собой ссылку. Во время присваивания ссылочных переменных объекты, на которые они ссылаются, не копируются – дублируются лишь сами ссылки. Для получения копий объектов применяется ключевое слово *clone*.

Однако для некоторых классов необходимо выполнить дополнительную работу, например, изменить значения некоторых свойств автоматически, сразу же после клонирования. Применяется специальный метод *\_\_clone*(), который автоматически вызывается *PHP* при клонировании объектов. В момент вызова метода *\_\_clone*() данные объекта уже скопированы в $*this* побитно. Достаточно изменить (или удалить) только нужные свойства, не трогая все остальные.

Одна из особенностей определения собственного метода *clone*() заключается в том, что его можно объявить закрытым (*private*). В этом случае в программе нельзя будет создать копию объекта никакими способами. В некоторых ситуациях это может оказаться полезным – существуют объекты, для которых операция клонирования бессмысленна, и ее нужно запретить.

Пример 10.10. Класс с методом \_\_*clone.*

//Создаем простой класс "Студент университета"

class Student

{

//Свойства имени, фамилии и возраста

public $name, $surname, $age;

//Конструктор инициализирует свойства

public function \_\_construct($name, $surname, $age)

{

$this->surname = $surname;

$this->name = $name;

$this->age = $age;

}

//Метод обнуляет значение свойств клона

public function \_\_clone()

{

$this->surname = 'Неизвестно...';

$this->name = 'Неизвестно...';

$this->age = 'Неизвестно...';

}

}

//Создаем новый объект студента, заполняем его

//свойства, а затем клонируем

$student = new Student('Денис', 'Демидов', 20);

$clone = clone $student;

### 10.1.5. Сериализация объектов

Функция *serialize*() возвращает строковое представление любого значения, которое может быть сохранено в *PHP*. Функция *unserialize*() может использовать эту строку для восcтановления исходного значения переменной. Использование сериализации для сохранения объекта сохранит все его переменные. Методы в объекте не будут сохранены, сохраняется только имя класса. Поэтому, для десериализации в программе должен быть уже объявлен соответствующий класс.

Пример 10.11. Сериализация объекта.

class A

{

public $one = 1;

public function show\_one()

{

echo $this->one;

}

}

//сериализуем объект класса

$a = new A;

$s = serialize($a);

//десериализуем строку в объект

$a = unserialize($s);

$a->show\_one();

## 10.2. Наследование

Наследование позволяет нам расширять функциональность классов, создавая на основе одних классов другие. Обычно идет речь о двух видах расширения функциональности – включении одних классов внутрь других или же о добавлении (изменении) в имеющиеся классы новых свойств и методов.

### 10.2.1. Включение

При включении объект определенного ранее класса помещается в свойство нового класса.

Пример 10.12. Класс с включением.

class A

{

function testA() { ... }

function test() { ... }

}

class B

{

var $a; // объект класса A

function \_\_construct(параметры\_для\_A, другие\_параметры) {

$a = new A(параметры\_для\_A);

// инициализируем другие поля B

}

function testB() { ... }

function test() { ... }

}

В примере 10.12 реализация объекта класса *B* содержит в своем составе подобъект класса *A* в качестве свойства. Это свойство – лишь часть объекта класса *B*, не более того. Подобъект не знает, что он в действительности не самостоятелен, а содержится в классе *B*.

### **10.2.2. Расширение класса**

Обычно под наследованием понимают именно расширение класса таким образом, что новый объект получает новые свойства и методы.

Пример 10.13. Расширение класса.

class B extends A

{

function \_\_construct(параметры\_для\_A, другие\_параметры) {

$this->A(параметры\_для\_A);

// инициализируем другие поля B

}

function testB() { ... }

function test() { ... }

}

На практике это означает, что везде, где допустима работа с объектами класса *A*, была допустима и работа с объектами класса *B*. Но в предыдущем примере это было совсем не так.

Ключевое слово *extends* говорит о том, что создаваемый класс является расширением класса *A*. То есть *B* содержит те же самые свойства и методы, что и *A*, но, помимо них и еще некоторые дополнительные. Родительский класс *A* принято называть базовым классом, а класс дочерний класс *B* – производным от *A*.

При этом, если в производном классе объявляется метод с таким же названием, как в исходном, то происходит переопределение метода. Переопределенный метод может быть написан с нуля. Существует также возможность использования кода родительской функции (или любых других методов родительского класса) – в этом случае можно обратиться к родительскому классу через ключевое слово *parent*. Например, вызвать функцию: *parent*->*testA*()

### 10.2.3. Запрет наследования

При написании метода возможно явно запретить его переопределение в производных классах, используя модификатор *final*.

Пример 10.14. Метод с запретом наследования.

class Base

{

public final function test(){}

}

class Derive extends Base

{

public function test() {} // Ошибка! Нельзя переопределить!

}

### 10.2.4. Позднее статическое связывание

Новая функциональность позднего статического связывания, позволяет унаследованным методам иметь доступ к статическим константам, методам и свойствам класса потомка, а не только родительского класса.

Пример 10.15. Доступ к статическим свойствам производных классов.

class Name

{

static $name = 'Sergey';

public static function getName()

{

return self::$name;

}

public static function getStaticName() {

return static::$name;

}

}

class Name1 extends Name {

static $name = 'Ivan';

}

//выводит Sergey

print Name1::getName();

//выводит Ivan

print Name1::getStaticName();

Новое ключевое слово *static* указывает, что необходимо использовать константу унаследованного класса, вместо константы которая была определена в классе, где объявлен метод *getStaticName*(). Ключевое слово *static* было добавлено, чтобы реализовать новый функционал, а для обратной совместимости *self* работает также как и в предыдущих версиях *PHP*.

## 10.3. Полиморфизм

Полиморфизм (многоформенность) является следствием идеи наследования. Полиморфность класса – это свойство базового класса использовать функции производных классов, даже если на момент определения еще неизвестно, какой именно класс будет включать его в качестве базового и, тем самым, становиться от него производным.

Например, базовый класс имеет функцию *Draw*(), которая заставляет объект нарисовать самого себя. Все производные классы-фигуры переопределяют эту функцию (ведь каждую фигуру нужно рисовать по-особому). Также есть массив фигур (любых). Используя полиморфизм, возможно перебрать все элементы массива и вызвать для каждого из них метод *Draw*() – фигура сама определит, какого она типа и как ее рисовать.

Такой способ описания класса, когда до конца неизвестно, что именно он будет делать в программе, называют *абстрагированием*, а сам класс – *абстрактным*.

### 10.3.1. Абстрактные методы и классы

Функция, переопределяемая в производном классе, называется *виртуальной*. Механизм виртуальных функций позволяет, например, предоставлять функциям, ожидающим объект одного класса, объект другого, производного, класса. Еще один классический пример – класс, воплощающий собой свойства геометрической фигуры, и несколько производных от него классов – квадрат, круг, треугольник и т. д.

*Виртуальный (абстрактный) метод* нельзя вызвать, если он не был переопределен в производном классе.

Любой класс, содержащий хотя бы один абстрактный метод, сам является абстрактным Объект абстрактного класса невозможно создать. Специально для того, чтобы автоматически учесть эти особенности, в *PHP* предусмотрено ключевое слово – модификатор *abstract*. Можно объявить класс или метод как *abstract*, и тогда контроль за их некорректным использованием возьмет на себя сам *PHP*. Абстрактные классы можно использовать только для одной цели: создавать от них производные.

Пример 10.16. Абстрактный класс.

abstract class Page

{

protected $expires;

protected $store;

abstract public function id($name);

}

При объявлении абстрактного метода (например, *id*()) не нужно определять его тело – необходимо лишь поставить точку с запятой после его прототипа.

### 10.3.2. Интерфейсы

Интерфейс в *PHP* – это список методов, которые должны быть реализованы в наследующем интерфейс классе. Для создания интерфейса в *PHP* используют слово *interface*. Интерфейсы очень похожи на абстрактные классы в *PHP*, но в отличии от них интерфейсы не содержат методов с телами, а представляют собой только список методов, который должен быть реальзован в классе, реализующем этот интерфейс.

В интерфейсах все методы фактически являются абстрактными, при этом ключевое слово *abstract* при задании методов в интерфейсе не используется. Интерфейсы в *PHP* позволяют делать множественную реализацию, то есть один класс может реализовывать несколько интерфейсов.

Интерфейсы, так же как и обычные классы, могут наследовать друг друга. Для наследования в интерфейсах, так же как и в классах, используется ключевое слово *extends*.

Пример 10.17. Создание интерфейса.

// Набор тегов, которым снабжается страница

interface Tag

{

public function tags(); // возвращает теги

}

class News implements Tag

{

private $id;

public function tags()

{

//полезный код

}

}

В случае если класс подключает к себе интерфейсы, но реализует не все методы в них, он автоматически становится абстрактным.

Пример 10.18. Подключение интерфейса.

interface I

{

public function F();

public function G();

}

abstract class C implements I

{

public function F(){ echo "Hello"; };

}

Если пропустить ключевое слово *abstract* в описании класса *C*, то возникнет ошибка: *Fatal error: Class C contains 1 abstract methods and must therefore be declared abstract (I::F)*.

### 10.3.3. Трейты

В отличие от интерфейсов, трейты содержат не абстрактные методы, а общие фрагменты классов. Если заставить все классы реализовывать методы от нескольких интерфейсов, среди реализаций будет довольно много повторяющегося кода. Для решения этой проблемы как раз и предназначены трейты. Можно реализовать общую функциональность и подмешивать ее в любой класс, в котором данная функциональность может потребоваться.

Объявляются трейты при помощи ключевого слова *trait*, после которого следует название трейта и в фигурных скобках его содержимое. Для включения одного или нескольких трейтов в класс используется ключевое слово *use*.

Пример 10.19. Использование трейта.

trait Author

{

public function tags() { echo "Author::tags"; }

public function authors() { echo "Author::authors"; }

}

class News

{

use Author;

public function authors() { echo "News::authors"; }

}

$news = new News();

$news->tags();

Трейты перегружают методы базового класса, а методы текущего класса перегружают методы трейтов.

Если же в двух трейтах будет определен метод с одним и тем же именем, возникнет конфликт. Его можно разрешить, явно указав, какой из методов следует использовать в фигурных скобках после оператора *use*. Внутри фигурных скобок можно применять ключевые слова *insteadof* для указания, какой из методов следует использовать. Кроме этого, допускается использование ключевого слова *as* для указания нового псевдонима для конфликтующего метода.

## 10.4. Пространства имен

Пусть написана функция *length*(), вычисляющая количество элементов массива, и она используется в программе. Через некоторое время необходимо подключить библиотеку стороннего разработчика и в ней тоже есть функция *length*(), но уже для определения длины строки. Возникает конфликт имен.

Одно из решений – добавлять в имена функций некоторый префикс, отвечающий их назначению. Так появились *array\_keys*(), *array\_merge*(), *array\_splice*() и т. д. Такой подход гарантирует в определенной степени уникальность имени, но сильно удлиняет название функции. Применение префикса обладает и еще одним недостатком. Если нужно его поменять, то придется изменить в программе каждое имя функции и переменной. Кроме того, для решения проблемы уникальности имени многие разработчики *PHP* и популярных фреймворков стали использовать очень длинные имена классов, например, *Zend\_Cloud\_StorageService\_Adapter\_S3*.

Вместо использования префиксов, непосредственно в именах функций, можно их вынести "на уровень выше", поместив все объекты программы в так называемое *пространство имен*. Пространство имен – это имеющий имя фрагмент программы, содержащий в себе функции, переменные, константы и другие именованные сущности. Для того чтобы объявить пространство имен, используется ключевое слово *namespace*, после которого следует имя пространства.

Пример 10.20. Пространство имен.

namespace PHP7;

// Отладочная функция

function debug($obj)

{

echo "<pre>";

print\_r($obj);

echo "</pre>";

}

// Класс страницы

class Page

{

protected $title;

protected $content;

// Конструктор класса

public function \_\_construct($title = '', $content = '')

{ $this->title = $title; $this->content = $content; }

}

Оператор *namespace* должен располагаться в файле первым, до любых объявлений. В одном файле не рекомендуется использовать несколько пространств имен.

Пространство имен очень напоминает файловую систему. Используя слеш, можно добавлять произвольное количество подуровней. За счет этого класс с длинным именем *Zend\_Cloud\_StorageService\_Adapter\_S3* при помощи пространства имен *Zend*\*Cloud*\*StorageService*\*Adapter* может быть преобразован в класс с коротким названием *S3*.

Мы объявляем пространство *PHP7*, в связи с чем вместо полного имени *PHP7*\*classes*\*Page* можем использовать относительное имя *classes*\*Page*.

Так же можно указать абсолютное имя, которое начинается с ведущего слеша: \*PHP7*\*classes*\*Page*. В файлах, где объявлено пространство имен, для обращения к стандартным функциям, например к строковой функции *strlen*(), потребуется воспользоваться абсолютным именем \*strlen*(), чтобы сообщить *PHP*, что *strlen*() является функцией глобального пространства имен, а не \*PHP7.*

Объявление пространства имен в начале файла при помощи ключевого слова *namespace* помогает сократить имена только из данного пространства имен. В реальности приходится работать с несколькими различными пространствами. Здесь помогает механизм импортирования, который при помощи ключевого слова *use* позволяет создавать псевдонимы.

Пример 10.21. Импортирование.

require\_once 'few.php';

use PHP7\classes\Page as Page;

use PHP7\functions as functions;

$page = new Page('Контакты', 'Содержимое страницы');

functions\debug($page);

Допускается создание псевдонимов как для отдельных элементов пространства имен (класс *Page*), так и для подпространств (*functions*).

# Список литературы

1. Котеров Д. В. *PHP* 7 / Д.В.Котеров, И. В. Симдянов. – СПб.: БХВ-Петербург, 2017.- 1088 с.: ил.

2. Прохоренок Н.А. HTML, Javascript, PHP и MySQL. Джентльменский набор Web-мастера. — 4-е изд. Перераб. И доп. / Н.А. Прохоренок, В.А. Дронов. — СПб.: БЧВ-Петербург, 2015. — 768 с.: ил. — (Профессиональное программирование). ISBN: 978-5-9775-3130-6.

3. Веллинг Л. Разработка веб-приложений с помощью PHP и MySQL [Текст] / Л. Веллинг, Л. Томсон ; пер. с англ. А. А. Моргунова. — 4-е изд. — М.; СПб.; Киев : Вильямс, 2013. — 847 с. : рис. — ISBN 978-5-8459-1574-0 (рус.) (в пер.). — ISBN 978-0-672-32916-6 (анг.).

4. Котеров, Д. В. PHP 5 [Текст] / Д. В. Котеров, А. Ф. Костарев. — 2-е изд. — СПб. : БХВ-Петербург, 2016. — 1078 с. : рис. — (В подлиннике). — ISBN 978-5-9775-0315-0 (в пер.).

5. PHP: Hypertext Preprocessor[Электронный ресурс]. / The PHP Group. *URL: http://php.net/manual/ru/* (дата обращения: 26.11.2019).

Оглавление

[Предисловие 3](#_Toc25830788)

[Глава 1. Введение в PHP 5](#_Toc25830789)

[1.1. Характеристика языка PHP 5](#_Toc25830790)

[1.2. Стандарты оформления кода PSR 6](#_Toc25830791)

[1.3. Компоновка программы 12](#_Toc25830792)

[1.4. Базовые синтаксические конструкции 14](#_Toc25830793)

[1.5. Операторы изменения хода выполнения программы 26](#_Toc25830794)

[Глава 2. Использование функций в PHP 33](#_Toc25830795)

[2.1. Функции в PHP 33](#_Toc25830796)

[2.2. Области видимости переменных, глобальные переменные 38](#_Toc25830797)

[2.3. Передача параметров по ссылке, по значению, предопределённые параметры 40](#_Toc25830798)

[2.4. Рекурсия 41](#_Toc25830799)

[Глава 3. Массивы 43](#_Toc25830800)

[3.1. Ассоциативный массив и список 43](#_Toc25830801)

[3.2. Операции над массивами 46](#_Toc25830802)

[3.3. Изменение порядка элементов в массиве 53](#_Toc25830803)

[3.3.3. Пользовательская сортировка: usort, uasort, uksort 55](#_Toc25830804)

[3.4. Работа с массивами как с множествами 58](#_Toc25830805)

[3.5. Функции для работы с массивами 62](#_Toc25830806)

[Глава 4. Строки 65](#_Toc25830807)

[4.1. Кодировки 65](#_Toc25830808)

[4.2. Операции со строками 67](#_Toc25830809)

[Глава 5. Математические функции 77](#_Toc25830810)

[5.1. Округление чисел 77](#_Toc25830811)

[5.2. Генерация случайных чисел 78](#_Toc25830812)

[5.3. Преобразования между системами счисления 79](#_Toc25830813)

[5.4. Поиск минимума и максимума 80](#_Toc25830814)

[5.5. Не-числа 80](#_Toc25830815)

[5.6. Степенные функции 81](#_Toc25830816)

[5.7. Тригонометрические функции 81](#_Toc25830817)

[5.8. Сводная таблица математических функций 82](#_Toc25830818)

[Глава 6. Работа с файловой системой 84](#_Toc25830819)

[6.1. Работа с файлами и каталогами 84](#_Toc25830820)

[6.2. Блокировки 94](#_Toc25830821)

[6.3. Управление правами доступа 96](#_Toc25830822)

[6.4. Взаимодействие с внешними ресурсами 100](#_Toc25830823)

[Глава 7. Регулярные выражения в формате PCRE 103](#_Toc25830824)

[7.1. Синтаксис языка PCRE 103](#_Toc25830825)

[7.2. Функции для работы с регулярными выражениями 107](#_Toc25830826)

[7.3. Карманы 110](#_Toc25830827)

[7.4. Приемы работы с регулярными выражениями 112](#_Toc25830828)

[Глава 8. Работа с датой и временем в *PHP* 116](#_Toc25830829)

[8.1. Функции времени в *PHP* 116](#_Toc25830830)

[8.2. Календарные классы в PHP 121](#_Toc25830831)

[Глава 9. Расширенные возможности языка PHP 124](#_Toc25830832)

[9.1. Взаимодействие с внешними программами из *PHP* 124](#_Toc25830833)

[9.2. Управление интерпретатором PHP 128](#_Toc25830834)

[9.3. Генерация кода во время выполнения программы 130](#_Toc25830835)

[Глава 10. Объектно-ориентированное программирование в PHP 132](#_Toc25830836)

[10.1. Классы 132](#_Toc25830837)

[10.2. Наследование 140](#_Toc25830838)

[10.3. Полиморфизм 142](#_Toc25830839)

[10.4. Пространства имен 145](#_Toc25830840)

[Список литературы 148](#_Toc25830841)