Установка программного обеспечения

1. Скачиваем Denwer: https://yadi.sk/d/_e603skdVvTVmw

2. При установке ничего менять не нужно. Режим запуска лучше выбрать «**2**», чтобы виртуальный диск не создавался каждый раз при загрузке системы.

3. Запускаем: ярлык «Start Denwer» на рабочем столе.

4. Находим рабочую папку: **Z:\home\localhost\www**. Все документы нужно будет сохранять именно в этой папке.

5. Проверяем работоспособность.

В рабочей папке, с помощью Блокнота, а еще лучше **Notepad++**, создайте файл **test.php** со следующим кодом:

```
<html>
 <head>
  <meta charset="utf-8">
  <title>Тестовый пример</title>
 </head>
 <body>
<?
   $greeting = 'Привет';
   $name = 'Вася';
   $message = "$greeting, $name!";
   echo $message;
?>
</body>
</html>
```

Теперь откройте любой браузер и в адресной строке наберите: localhost/test.php

Если появилось: «**Привет, Вася!**», значит все работает.

Для набора кода советую в дальнейшем использовать **Notepad++**, либо ему подобные редакторы с подсветкой кода.

В случае возникновения проблем с кодировкой, установите кодировку самого файла **UTF-8 (Кодировки/Преобразовать в UTF-8)**.

Основы синтаксиса PHP

Как выглядит PHP-программа

PHP-программа представляет собой HTML-страницу со вставками кода.

PHP-скрипт (только HTML)

```
<html>
<head><title>Hello World</title></head>
<body><h1>Hello World!</h1></body>
</html>
```

Как видите, простейшая программа на PHP - это обычная HTML-страница.

Непосредственно PHP-код размещается между тэгами **<?** и **?>**. Все, что расположено между этими тэгами, заменяется на выведенный скриптом внутри этого блока HTML-кодом (в частном случае - если скрипт ничего не выводит - просто "исчезает").

> *Вообще, универсальный (то есть - гарантированно работающий при любой конфигурации PHP), но более длинный способ спецификации PHP-кода - тэги **<?php** ... **?>**. Такая длинная форма записи используется при совмещении XML и PHP, так как тэг **<?** ... **?>** используется в стандарте XML.*

Рассмотрим простой пример (номера строк уберите).

```
1. <html>
2. <head><title>Hello World</title></head>
3. <body><h1>Hello World!</h1>
4. <p>Текущая дата:
5. <?
6.   echo date("d.m.Y");
7. ?>
8. </body>
9. </html>
```

Если сегодня - 23-е сентября 2014 года, в результате исполнения скрипта браузер получит следующий HTML-код:

```
<html>
<head><title>Hello World</title></head>
<body><h1>Hello World!</h1>
<p>Текущая дата:
23.09.2014</body>
</html>
```

Строки 5,6,7 - вставка PHP-кода. На строках 5 и 7 расположены соответственно открывающий и закрывающий тэг. Их совершенно необязательно располагать на отдельных строках - это сделано по соображениям удобства чтения.

В строке 6 расположен оператор **echo**, используемый для вывода в браузер. Выводит же он результат выполнения функции **date** - в данном случае это текущая дата.

Строка 6 является законченным выражением. Каждое выражение в PHP заканчивается точкой с запятой - **;**.

Переменные и типы данных

Переменные в PHP начинаются со знака **$**, за которыми следует произвольный набор латинских букв, цифр и знака подчеркивания: **_**, при этом цифра не может следовать сразу за знаком **$**.

Регистр букв в имени переменной имеет значение: $A и $a - это две разные переменные.

Для присваивания переменной значения используется оператор **=**.

Пример:

```php
<?
  $a = 'test';
  $copyOf_a = $a;
  $Number100 = 100;
  echo $a;
  echo $copyOf_a;
  echo $Number100;
?>
```

Данный код выведет: **testtest100**.

В строке 2 переменной **$a** присваивается строковое значение **'test'**. Строки в PHP записываются в кавычках - одинарных или двойных (различие между записями в разных кавычках мы рассмотрим чуть позже). Также справедливо высказывание, что переменная **$a инициализируется** значением 'test': в PHP переменная создается при первом присваивании ей значения; если переменной не было присвоено значение - переменная **не определена**, то есть ее просто не существует.

В строке 3 переменная **$copyOf_a** инициализируется **значением переменной $a**; в данном случае (смотрим строку 2) это значение - строка 'test'. В строке с номером 4 переменной с именем **$Number100** присваивается числовое значение 100.

Как видите, в PHP существует типизация, то есть язык различает типы данных - строки, числа и т.д. Однако, при этом PHP является языком со **слабой типизацией** - преобразования между типами данных происходят автоматически по установленным правилам. Например, программа **<? echo '100' + 1; ?>** выведет число 101: строка автоматически преобразуется в число при использовании в числовом контексте (в данном случае - строка '100', при использовании в качестве слагаемого, преобразуется в число 100, так как операция сложения для строк не определена).

Рассмотрим еще один пример:

```php
<?
  $greeting = 'Привет';
  $name = 'Вася';
  $message = "$greeting, $name!";
  echo $message;
?>
```

Особого внимания заслуживает четвертая строка. Внутри двойных кавычек указаны переменные, определенные в предыдущих строках. Если выполнить эту программу (вы ведь это уже сделали? :)), в окне браузера отобразится строка **Привет, Вася!**. Собственно, в этом и заключается

основная особенность двойных кавычек: имена переменных, указанных внутри пары символов **"**, заменяются на соответствующие этим переменным значения.

Помимо этого, внутри двойных кавычек распознаются специальные управляющие комбинации, состоящие из двух символов, первый из которых - обратный слэш (****). Наиболее часто используются следующие управляющие символы:

- **\r** - возврат каретки (CR)
- **\n** - перевод строки (NL)
- **\"** - двойная кавычка
- **\$** - символ доллара ($)
- **** - собственно, обратный слэш (\)

Символы **\r** и **\n** обычно используются вместе, в виде комбинации **\r\n** - так обозначается перевод строки в Windows и многих TCP/IP-протоколах. В Unix новая строка обозначается одним символом **\n**; обычно такой способ перевода строки используется и в HTML-документах (конечно же, это влияет только на HTML-код, но не отображение в браузере (если только текст не заключен в пару тэгов <pre>...</pre>): для отображаемого перевода строки, как известно, используется тэг
).

Оставшиеся три пункта из приведенного списка применения обратного слэша являются примерами **экранирования** - отмены специального действия символа. Так, двойная кавычка обозначала бы конец строки, символ доллара - начало имени переменной, а обратный слэш - начало управляющей комбинации (о которых мы тут и говорим ;)). При экранировании, символ воспринимается "как он есть", и никаких специальных действий не производится.

Если в данном кавычки заменить на одинарные, в браузере отобразится именно то, что внутри них написано (**$greeting, $name!**). Комбинации символов, начинающиеся с ****, в одинарных кавычках также никак не преобразуются, за двумя исключениями: **\'** - одинарная кавычка внутри строки; **** - обратный слэш (в количестве одна штука :).

Немного изменим наш последний пример:

```
<?
  $greeting = 'Привет';
  $name = 'Вася';
  $message = $greeting . ',' . $name. '!';
  echo $message;
?>
```

На этот раз мы не стали пользоваться "услужливостью" двойных кавычек: в строке 4 имена переменных и строковые константы записаны через оператор **конкатенации** (объединения строк). В PHP конкатенация обозначается точкой - **.**. Результат выполнения этой программы аналогичен предыдущему примеру.

Я рекомендую использовать именно этот способ записи - на то есть достаточно причин:

- Имена переменных более четко визуально отделены от строковых значений, что лучше всего заметно в редакторе с подсветкой кода;
- Интерпретатор PHP обрабатывает такую запись немного быстрее;
- PHP сможет более четко отследить опечатку в имени переменной;
- Вы не совершите ошибку, подобную следующей: **$message = "$greetingVasya"** - PHP в данном случае выведет не "ПриветVasya", а пустую строку, ибо **$greetingVasya** распознается как имя переменной, а таковой у нас нет.

Однако, двойные кавычки весьма популярны, и вы их наверняка встретите во множестве скриптов, доступных в сети.

Помимо строк и чисел, существует еще один простой, но важный тип данных - **булевый** (bool), к которому относятся два специальных значения: **true** (истина) и **false** (ложь). При автоматическом приведении типов, false соответствует числу 0 и пустой строке (**''**), true - всему остальному. Булевые значения часто применяются совместно с условными операторами, о которых мы дальше и поговорим.

Условные операторы

if

Часто (да что тут говорить, практически в любой программе) возникает необходимость выполнения разного кода в зависимости от определенных условий. Рассмотрим пример:

```
1.  <?
2.    $i = 10;
3.    $j = 5 * 2;
4.    if ($i == $j)
5.      echo 'Переменные $i и $j имеют одинаковые значения';
6.    else
7.      echo 'Переменные $i и $j имеют различные значения';
8.  ?>
```

Здесь используется оператор if..else - **условный оператор**. В общем виде он выглядит так:
if (условие)
выражение_1;
else
выражение_2;

В данном случае, условием является результат сравнения значений переменных $i и $j. Оператор сравнения - **==** - два знака равенства. Поскольку 5*2 равняется 10, и, соответственно, 10 равняется 10 ;), выполнится строка 5, и мы увидим, что переменные имеют равные значения. Измените, например, строку 2 на **$i = 11**, и вы увидите, что выполнится оператор echo из строки 7 (так как условие ложно). Помимо **==**, есть и другие операторы сравнения:
!= - не равно;
< - меньше;
> - больше;
<= - меньше или равно;
>= - больше или равно.

Если требуется только выполнить действие, если условие выполняется, блок **else ...**можно опустить:

```
1.  <?
2.    $i = 10;
3.    $j = 5 * 2;
4.    if ($i == $j)
5.      echo 'Переменные $i и $j имеют одинаковые значения';
6.  ?>
```

В этом случае, если условие ложно, в браузер не выведется ничего.

Отступы перед строками **echo ...** сделаны для удобства чтения, но PHP они ни о чем не говорят. Следующий пример работает не так, как можно ожидать:

```
1.  <?
2.    $i = 10;
3.    $j = 11;
4.    if ($i > $j)
5.      $diff = $j - $i;
6.      echo '$j больше, чем $i; разность между $j и $i составляет ' . $diff; //НЕВЕРНО!
7.  ?>
```

Вопреки возможным ожиданиям, строка 6 выполнится, хотя условие ($i > $j) ложно. Дело в том, что к if(...) относится лишь следующее выражение - строка 5. Строка 6 же выполняется в любом случае - действие if(..) на нее уже не распространяется. Для получения нужного эффекта следует воспользоваться **блоком операторов**, который задается фигурными скобками:

```
1.  <?
2.    $i = 10;
3.    $j = 11;
4.    if ($i > $j) {
5.      $diff = $j - $i;
6.      echo '$j больше, чем $i; разность между $j и $i составляет ' . $diff;
7.    }
8.  ?>
```

Теперь все работает правильно.

Фигурные скобки можно использовать, даже если внутри - только один оператор. Я рекомендую поступать именно так - меньше шансов ошибиться. На производительности это никак не сказывается, зато повышает читабельность.

Часто нужно ввести дополнительные условия (если так... а если по-другому... иначе) или даже (если так.. а если по-другому.. а если еще по-другому... иначе):

```
1.  <?
2.    $i = 10;
3.    $j = 11;
4.    if ($i > $j) {
5.      echo '$i больше, чем $j';
6.    } else if ($i < $j) {
7.      echo '$i меньше, чем $j';
8.    } else {     // ничего, кроме равенства, не остается :)
9.      echo '$i равно $j';
10.   }
11. ?>
```

Для дополнительных "развилок" используется оператор **if... else if ... else**. Как и в случае с if, блок **else** может отсутствовать. Следуя свой же недавней рекомендации, я заключил все операторы echo в фигурные скобки, хотя все бы прекрасно работало и без оных.

Кстати, в строке 8 - комментарий. Это информация для человека, PHP ее игнорирует. Комментарии бывают двух видов: однострочный, как здесь - начинается с **//** и распространяется до конца строки, и многострочный - комментарием считается все, что расположено между парами символов **/*** и ***/**.

switch

Бывает необходимость осуществления "развилки" в зависимости от значения одной и той же переменной или выражения. Можно написать что-то вроде:

```
if ($i==1) {
  // код, соответствующий $i==1
} else if ($i==2) {
  // код, соответствующий $i==2
} else if ($i==3) {
  // код, соответствующий $i==3...
}
```

Но существует более удобный для этого случая оператор - **switch**. Выглядит это так:

```
1.  <?
2.    $i = 1;
3.
4.    switch ($i) {
5.      case 1:
6.        echo 'один';
7.        break;
8.      case 2:
9.        echo 'два';
10.       break;
11.case 3:
12.  echo 'три';
13.       break;
14.default:
15.  echo 'я умею считать только до трех! ;)';
16.   }
17. ?>
```

Понаблюдайте за результатом выполнения программы, меняя значение $i во второй строке. Как вы уже наверняка поняли, после **switch** в скобках указывается переменная (хотя там может быть и выражение - например, **$i+1** - попробуйте :)), а строки **case XXX** соответствуют значению того, что в скобках.

Операторы, находящиеся между case-ами, не нужно заключать в фигурные скобки - каждое ответвление заканчивается оператором **break**.

Специальное условие **default** соответствует "всему остальному" (аналог **else** в if...else if..else). **default** всегда располагется последним, так что break здесь необязателен. Как и в случае с **else**, условие **default** может отсутствовать.

Если вы вдруг забудете указать **break**, будут выполняться все последующие строки - из последующих **case**-ов! Например, если в нашем примере удалить строку 6, при $i==1 в браузер выведется "**одиндва**". Некоторые чересчур хитрые программисты используют этот трюк для указания нескольких вариантов значений:

```
1.  <?
2.    $i = 1;
3.
4.    switch ($i) {
5.      case 0: // break отсутствует умышленно!
6.      case 1:
7.        echo 'ноль или один';
8.        break;
9.      case 2:
10. echo 'два';
11. break;
12.case 3:
13.  echo 'три';
14.  break;
15.   }
16. ?>
```

или для выполнения при определенном значении условия двух действий подряд. Но это уже ухищрения - лучше всего использовать **switch** "как положено", заканчивая каждый case своим break-ом; а если уж "ухищряетесь" - не забудьте поставить комментарий, как это сделано в строке 5 последнего примера.

Циклы

Любой более-менее серьезный язык программирования содержит операторы организации циклов для повторного выполнения фрагментов кода. В PHP есть три таких оператора.

while

Начнем с цикла while:

```
1.  <?
2.    $i = 1;
3.    while($i < 10) {
4.      echo $i . "<br>\n";
5.      $i++;
6.    }
7.  ?>
8.
```

Цикл **while** (строка 3) работает следующим образом. Сначала проверяется истинность выражения в скобках. Если оно не истинно, тело цикла (все, что расположено между последующими фигурными скобками - или, если их нет - следующая инструкция) не выполняется. Если же оно истинно, после выполнения кода, находящегося в теле цикла, опять проверяется истинность выражения, и т.д.

В теле цикла (строки 4,5) выводится текущее значение переменной $i, после чего значение $i увеличивается на единицу.

Переменную, используемую подобно **$i** в данном примере, часто называют переменной-счетчиком цикла, или просто счетчиком.

***$i++**, операция **инкрементирования** (увеличения значения на 1) - сокращенная запись для **$i=$i+1**; аналогичная сокращенная запись - **$i+=1**. По последнему правилу можно сокращать любые бинарные операции (например, конкатенация: **$s .= 'foo'** - аналог **$s = $s . 'foo'**); однако, аналогично инкрементированию можно записать только **декрементирование** (уменьшение значения на 1): $i--.*

*Возможна также запись ++$i (и --$i); различие в расположении знаков операции проявляется только при непосредственном использовании результата этого вычисления: если $i равна 1, в случае **$j=$i++** переменная **$j** получит значение 1, если же **$j=++$i**, **$j** будет равняться двум. Из-за этой особенности операция **++$i** называется **преинкрементом**, а **$i++** - постинкрементом.*

Если бы мы не увеличивали значение **$i**, выход из цикла никогда бы не произошел ("вечный цикл").

Запишем тот же пример в более краткой форме:

```
<?
 $i = 1;
 while($i < 10) {
  echo $i++ . "<br>\n";
 }
?>
```

И еще один вариант:

```
<?
 $i = 0;
 while(++$i < 10) {
  echo $i . "<br>\n";
 }
?>
```

Советую немного поразмыслить, почему все эти три программы работают одинаково. Заметьте, что в зависимости от начального значения счетчика удобнее та или иная форма записи.

do..while

Цикл **do..while** практически аналогичен циклу while, отличаясь от него тем, что условие находится в конце цикла. Таким образом, тело цикла do..while выполняется хотя бы один раз.

Пример:

```
<?
 $i = 1;
 do {
  echo $i . "<br>\n";
 } while ($i++ < 10);
?>
```

for

Цикл **for** - достаточно универсальная конструкция. Он может выглядеть как просто, так и очень запутанно. Рассмотрим для начала классический вариант его использования:

```
<?
 for ($i=1; $i<10; $i++) {
  echo $i . "<br>\n";
 }
?>
```

Как и в предыдущих примерах, этот скрипт выводит в браузер числа от 1 до 9. Синтаксис цикла **for** в общем случае такой:

for(выражение_1;выражение_2;выражение_3), где **выражение_1** выполняется перед выполнением цикла,**выражение_2** - условие выполнения цикла (аналогично while), а **выражение_3** выполняется после каждой итерации цикла.

Операторы break и continue. Вложенные циклы

Может вознкнуть необходимость выхода из цикла при определенном условии, проверяемом в теле цикла. Для этого служит оператор break, с которым мы уже встречались, рассматривая **switch**.

```
<?
 $i = 0;
 while (++$i < 10) {
  echo $i . "<br>\n";
  if ($i == 5) break;
 }
?>
```

Этот цикл выведет только значения от 1 до 5. При $i==5 сработает условный оператор **if** в строке 5, и выполнение цикла прекратится.

Оператор **continue** начинает новую итерацию цикла. В следующем примере с помощью **continue** "пропускается" вывод числа 5:

```
<?
 for ($i=0; $i<10; $i++) {
  if ($i == 5) continue;
  echo $i . "<br>\n";
```

```
  }
?>
```

Операторы break и continue можно использовать совместно со всеми видами циклов.

Циклы могут быть вложенными (как практически все в PHP): внутри одного цикла может располагаться другой цикл, и т.д. Операторы break и continue имеют необязательный числовой параметр, указывающий, к какому по порядку вложенности циклу - считая снизу вверх от текущей позиции - они относятся (на самом деле, **break** - это сокращенная запись **break 1** - аналогично и с **continue**). Пример выхода из двух циклов сразу:

```
<?
  for ($i=0; $i<10; $i++) {
    for ($j=0; $j<10; $j++) {
      if ($j == 5) break 2;
      echo '$i=' . $i . ', $j=' . $j . "<br>\n";
    }
  }
?>
```

Массивы

Массив представляет собой набор переменных, объединенных одним именем. Каждое значение массива идентифицируется **индексом**, который указывается после имени переменной-массива в квадратных скобках. Комбинацию индекса и соответствующего ему значения называют элементом массива.

```
<?
  $i = 1024;
  $a[1] = 'abc';
  $a[2] = 100;
  $a['test'] = $i - $a[2];

  echo $a[1] . "<br>\n";
  echo $a[2] . "<br>\n";
  echo $a['test'] . "<br>\n";
?>
```

В приведенном примере, в строке три объявляется **элемент массива `$a`** с **индексом `1`**; элементу массива присваивается строковое значение 'abc'. Этой же строкой объявляется и массив $a, так как это первое упоминание переменной $a в контексте массива, массив создается автоматически. В строке 4 элементу массива с индексом 2 присваивается числовое значение 100. В строке же 5 значение, равное разности $i и $a[2], присваивается элементу массива $a со **строковым** индексом 'test'.

Как видите, индекс массива может быть как числом, так и строкой.

В предыдущем примере массив создавался автоматически при описании первого элемента массива. Но массив можно задать и явно:

```
<?
  $i = 1024;
  $a = array( 1=>'abc', 2=>100, 'test'=>$i-100 );
  print_r($a);
?>
```

Созданный в последнем примере массив $a полностью аналогичен массиву из предыдущего примера. Каждый элемент массива здесь задается в виде *индекс*`=>`*значение*. При создании элемента 'test' пришлось указать значение 100 непосредственно, так как на этот раз мы создаем массив "одним махом", и значения его элементов на этапе создания неизвестны PHP.

В строке 4 для вывода значения массива мы воспользовались функцией **print_r()**, которая очень удобна для вывода содержимого массивов на экран - прежде всего, в целях отладки.

Если явно не указывать индексы, то здесь проявляется свойство массивов PHP, характерное для числовых массивов в других языках: очередной элемент будет иметь порядковый числовой индекс. Нумерация начинается с нуля. Пример:

```
<?
  $operating_systems = array( 'Windows', 'Linux', 'FreeBSD', 'OS/2');
  $operating_systems[] = 'MS-DOS';

```

```
5.    echo "<pre>";
6.    print_r($operating_systems);
7.    echo "</pre>";
8.  ?>
```

Вывод:

```
Array
(
   [0] => Windows
   [1] => Linux
   [2] => FreeBSD
   [3] => OS/2
   [4] => MS-DOS
)
```

Здесь мы явно не указывали индексы: PHP автоматически присвоил числовые индексы, начиная с нуля. При использовании такой формы записи массив можно перебирать с помощью цикла for. Количество элементов массива возвращает оператор **count** (или его синоним, **sizeof**):

```
1.  <?
2.    $operating_systems = array( 'Windows', 'Linux', 'FreeBSD', 'OS/2');
3.    $operating_systems[] = 'MS-DOS';
4.
5.    echo '<table border=1>';
6.    for ($i=0; $i<count($operating_systems); $i++) {
7.      echo '<tr><td>' . $i . '</td><td>' . $operating_systems[$i] . '</td></tr>';
8.    }
9.    echo '</table>';
10. ?>
```

Стили записи можно смешивать. Обратите внимание на то, какие индексы автоматически присваиваются PHP после установки некоторых индексов вручную.

```
1.  <?
2.    $languages = array(
3.      1 => 'Assembler',
4.      'C++',
5.      'Pascal',
6.      'scripting' => 'bash'
7.    );
8.    $languages['php'] = 'PHP';
9.    $languages[100] = 'Java';
10.   $languages[] = 'Perl';
11.
12.   echo "<pre>";
13.   print_r($languages);
14.   echo "</pre>";
15. ?>
```

Вывод:

```
Array
(
   [1] => Assembler
   [2] => C++
   [3] => Pascal
   [scripting] => bash
   [php] => PHP
   [100] => Java
   [101] => Perl
)
```

Цикл foreach

Массив, подобный предыдущему, перебрать с помощью for затруднительно. Для перебора элементов массива предусмотрен специальный цикл **foreach**:

```
1.  <?
2.    $languages = array(
3.      1 => 'Assembler',
4.      'C++',
```

```
5.      'Pascal',
6.      'scripting' => 'bash'
7.    );
8.    $languages['php'] = 'PHP';
9.    $languages[100] = 'Java';
10.   $languages[] = 'Perl';
11. ?>
12. <table>
13. <tr>
14.   <th>Индекс</th>
15.   <th>Значение</th>
16. </tr>
17. <?
18.   foreach ($languages as $key => $value) {
19.     echo '<tr><td>' . $key . '</td><td>' . $value . '</td></tr>';
20.   }
21. ?>
22. </table>
```

Этот цикл работает следующим образом: в порядке появления в коде программы элементов массива **$languages**, переменным **$key** и **$value** присваиваются соответственно индекс и значение очередного элемента, и выполняется тело цикла.

Если индексы нас не интересуют, цикл можно записать следующим образом: **foreach ($languages as $value)**.

Конструкции list и each

В дополнение к уже рассмотренной конструкции **array**, существует дополняющая ее конструкция **list**, являющаяся своего рода антиподом **array**: если последняя используется для создания массива из набора значений, то **list**, напротив, заполняет перечисленные переменные значениями из массива.

Допустим, у нас есть массив **$lang = array('php', 'perl', 'basic')**. Тогда конструкция **list($a, $b) = $lang** присвоит переменной $a значение 'php', а $b - 'perl'. Соответственно, **list($a, $b, $c) = $lang** дополнительно присвоит $c = 'basic'.

*Если бы в массиве **$lang** был только один элемент, PHP бы выдал замечание об отсутствии второго элемента массива.*

А если нас интересуют не только значения, но и индексы? Воспользуемся конструкцией each, которая возвращает пары индекс-значение.

```
1.  <?
2.    $browsers = array(
3.      'MSIE' => 'Microsoft Internet Explorer 6.0',
4.      'Gecko' => 'Mozilla Firefox 0.9',
5.      'Opera' => 'Opera 7.50'
6.    );
7.
8.    list($a, $b) = each($browsers);
9.    list($c, $d) = each($browsers);
10. list($e, $f) = each($browsers);
11. echo $a.':'.$b."<br>\n";
12. echo $c.':'.$d."<br>\n";
13. echo $e.':'.$f."<br>\n";
14. ?>
```

На первый взгляд может удивить тот факт, что в строках 8-10 переменным присваиваются разные значения, хотя выражения справа от знака присваивания совершенно одинаковые. Дело в том, что у каждого массива есть скрытый указатель текущего элемента. Изначально он указывает на первый элемент. Конструкция each же продвигает указатель на один элемент вперед.

Эта особенность позволяет перебирать массив с помощью обычных циклов while и for. Конечно, ранее рассмотренный цикл **foreach** удобнее, и стоит предпочесть его, но конструкция с использованием **each** довольно распространена, и вы можете ее встретить во множестве скриптов в сети.

```
1.  <?
2.    $browsers = array(
3.      'MSIE' => 'Microsoft Internet Explorer 6.0',
```

```
    'Gecko' => 'Mozilla Firefox 0.9',
    'Opera' => 'Opera 7.50'
  );

  while (list($key,$value)=each($browsers)) {
    echo $key . ':' . $value . "<br>\n";
  }

?>
```

После завершения цикла, указатель текущего элемента указывает на конец массива. Если цикл необходимо выполнить несколько раз, указатель надо принудительно сбросить с помощью оператора **reset**: **reset($browsers)**. Этот оператор устанавливает указатель текущего элемента в начало массива.

Константы

В отличие от переменных, значение константы устанавливается единожды и не подлежит изменению. Константы не начинаются с символа **$** и определяются с помощью оператора **define**:

```
<?
  define ('MY_NAME', 'Вася');

  echo 'Меня зовут ' . MY_NAME;
?>
```

Константы необязательно называть прописными буквами, но это общепринятое (и удобное) соглашение.

Поскольку имя константы не начинается с какого-либо спецсимвола, внутри двойных кавычек значение константы поместить невозможно (так как нет возможности различить, где имя константы, а где - просто текст).