**Возможности РНР**

В основном **РНР** используется в связке ***с базами данных***. На данный момент они являются самыми удобными хранилищами информации. Сейчас даже самые обычные гостевые книги делают с применением баз данных, а большие информационные порталы вообще сложно представить без них. Язык **РНР** позволяет работать с различными типами баз данных, такими как **MySQL, ODBC, Oracle** (**OCI7** и **OCI8**), **InterBase** и др.

Также широко используется возможность работы с электронной почтой. Поддерживаются такие протоколы, как**IMAP, РОРЗ, SMTP** и другие для отправки и приема сообщений. Поэтому очень просто создать **РНР**-приложение для рассылки писем.

Ранее говорилось о том, что с помощью **РНР** создаются **Web**-страницы, но его возможности на этом не исчерпываются. С помощью **РНР** можно создавать изображения, файлы типа **PDF**, а также **Flash**-анимации.

Работа с файлами тоже является частью рассматриваемого языка. **РНР** представляет полный спектр функций для этого: вы сможете удалять, копировать, создавать файлы, записывать и извлекать из них нужную информацию.

**Необходимое программное обеспечение**

Необходимое программное обеспечение позволит вам работать с **PHP** на своем компьютере. Для этого вам понадобится **Web**-сервер и модули **РНР**.

Для серверного программирования требуется удаленный компьютер. Но при таких условиях создавать и тестировать **PHP**-приложения неудобно, так как для этого надо постоянно поддерживать связь с сервером. При отсутствии соединения работа вообще не представляется возможной. Поэтому **Web**-сервер и модули **РНР** лучше установить на свой компьютер. При этом он будет являться одновременно клиентом и сервером.

**Apache.**

Многие ведущие программисты используют один из лучших **Web**-серверов - **Apache**, разработчики которого предлагают как версии под **Windows**, так и для **UNIX**-подобных систем. **Apache** завоевал огромную популярность в сети **Internet**: сейчас под ним работает до 67% **Web**-сайтов, в то время как **IIS** применяется лишь на 21% серверов.

**РНР.**

Что касается **РНР**, то создатели языка предлагают нам как последние стабильные версии, так и релизы. Принципиальной разницы в выборе версии **РНР** нет: вы можете изучить основы языка на **РНР 4**, а затем установить одну из последних версий и оценить новые возможности движка и стандартных функций. **РНР** и **Apache** работают на платформе **Windows**, но благодаря переносимости языка, вы сможете запускать созданные вами **PHP**-программы и в среде **UNIX**.

Описание установки **Apache** и **PHP** приведено соотвественно в приложениях [2](http://it.kgsu.ru/PHP/pril2.html) и [3](http://it.kgsu.ru/PHP/pril3.html). Можно также воспользоваться пакетом Денвер, описание которого дано в [приложении 1](http://it.kgsu.ru/PHP/pril1.html).

**MySQL.**

Мы уже говорили, что язык **РНР** позволяет программисту работать с множеством баз данных. Одной из наиболее популярных СУБД (Система управления базами данных) является **MySQL** **(SQL: Strutted Query Language -*Структурированный язык запросов*)**.

Умение устанавливать и настраивать **Apache, РНР** и **MySQL** является неотъемлемой частью работы программиста, создающего приложения для **Web**.

**Алфавит языка**

***Символы языка***. При составлении программы нужно знать, какие символы можно использовать. Это:

* большие и малые латинские буквы;
* большие и малые русские буквы, которые используются только для вывода сообщений на экран и создания комментариев;
* цифры от 0 до 9;
* символ **"$"**. С этого символа начинаются имена переменных;
* символ "\_" (подчеркивание). Он может использоваться при задании имен переменных;
* двадцать один специальный символ: + - \* / = > < . , ; : @ ' ( ) [ ] { } # ^.

Из этих символов можно составить:

* служебные (ключевые, зарезервированные) слова;
* имена;
* константы;
* знаки операций;
* разделители (знаки пунктуации).

***Слова***. Из отдельных символов языка образуются слова, которые имеют определенный смысл в программе. Слова в тексте разделяются пробелами или другими специальными символами. В зависимости от назначения различают ***служебные слова*** и ***имена***.

***Служебные слова***. Каждое служебное слово в **PHP** имеет заранее определенное назначение и его нельзя использовать для других целей. В тексте программы это могут быть: **return, define, echo, while** и т.д.

***Имена***. Имена - это слова, используемые автором программы (программистом) для обозначения каких-либо элементов. Именем может быть любая последовательность символов алфавита, начинающаяся с символа **"$"**, например: **$Elena, $Hello\_World** (помните, что символ подчеркивания - это буква). Имена могут быть любой длины. В качестве имен нельзя использовать служебные слова.

***Константа***. Константа - это фиксированное числовое, строковое или символьное значение.

**Основы синтаксиса PHP**

Для создания программ можно использовать обычный текстовый редактор. Например, ***блокнот*** **Windows**.

Напишем первую программу на PHP.

Для этого сначала необходимо запустить сервер.

Затем откройте блокнот и наберите следующий текст:

**<?php**

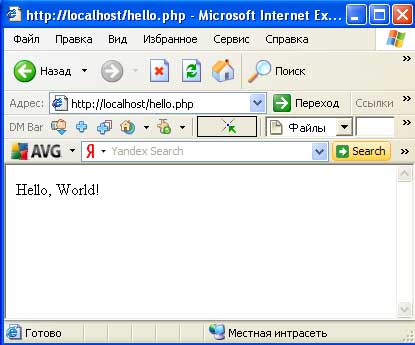
**echo "Hello, World!";**

**?>**

Сохраните этот файл под именем **hello** с расширением **.php** в корневой каталог вашего **Web**-сервера. Так как мы работаем с **Denwer**, то в нашем случае это **<*Имя диска*>:\WebServers\home\localhost\www**.

Затем откройте браузер и введите в адресной строке адрес страницы. В нашем случае это**http://localhost/hello.php**.

Результат выполнения программы вы можете увидеть на рисунке 1.

  
Рис.1. Результат работы программы

Рассмотрим текст этой программы. Каждая программа на **PHP** заключается в теги **<?php ... ?>**. Внутри данного тега заключается тело программы. С помощью этой конструкции выделяется **PHP**-код. Вне этого тега конструкции языка **PHP** выполняться не будут. Тег **echo** используется для вывода строки на экран. Подробнее этот тег мы рассмотрим позже. Каждая конструкция в **PHP** должна заканчиваться символом **";"**.

В меню браузера выберите пункт ***Вид* | *Просмотр* HTML-*кода***. **Web**-сервер передал браузеру только строку**"Hello, World!"**. Именно в этом состоит главная особенность серверного программирования, т.е. сначала выполняется на сервере **PHP**-код, а затем результат отправляется браузеру. В **PHP**-код можно так же вставлять**HTML**-теги.

В **PHP** можно использовать комментарии. Они бывают однострочные и многострочные.

Для однострочных комментариев используется тег **//**. Например:

**<?php**

**echo "Hello, World!"; *// на экран выведется "Hello, World!"***

**?>**

    Для многострочных комментариев используется тег **/\* \*/**. Например:

**<?php**

**echo "Hello, World!";**

***/\*на экран будет выведена строка***

***"Hello, World!"\*/***

**?>**

**Переменные и константы**

Теперь мы перейдем к изучению переменных и констант.

*Переменная* - средство языка для хранения данных.

Синтаксические правила для имен переменных следующие:

* названия всех переменных в PHP начинается со знака $;
* после него может идти любой символ, кроме цифры;
* далее могут идти цифры, буквы, знак подчеркивания в любой последовательности, но знак пробела недопустим;
* имена переменных чувствительны к регистру, то есть переменные с именами $Num и $num будут считаться различными.

Старайтесь выбирать имя переменной так, чтобы по нему можно было понять смысл хранимой информации.

Теперь рассмотрим типы данных в PHP.

Integer - множество целых чисел с определенными ограничениями по величине, например, на 32-битных платформах диапазон составляет от -2147483648 до 2147483647.

Double - множество вещественных чисел.

Boolean - логический тип данных, то есть множество, состоящее из двух значений: true (истина) и false (ложь).

String - строка, длина которой практически не ограничена.

Array - массив.

Object - объект.

Чтобы проверить, существует ли переменная с заданным именем, необходимо воспользоваться функцией isset().

  
Рис.1. Общий вид функции isset()

Эта функция устанавливает, определена ли переменная. Если переменная существует, то isset() возвращает 1, в противном случае - 0.

Например:

<?php

$a=12;

echo isset($a); ***// выведет 1***

echo isset($b); ***// ничего не выведет***

?>

Если были переданы несколько переменных, то isset() вернет 1 только в том случае, если все заданные переменные определены.

*Константа* - неизменяемое значение.

Общий вид определения констант:

  
Рис.2. Общий вид определения констант

Например:

<?php

define("name", "stark");

define("num", 12.68);

echo "Константа name = ";

echo name; ***// выводит stark***

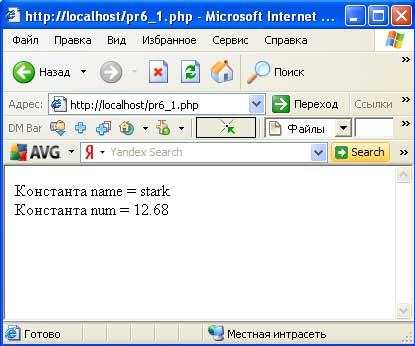
echo "<br>";

echo "Константа num = ";

echo num; ***// выводит 12.68***

?>

Результат выполнения рассмотренной программы вы можете увидеть на рисунке 3.

  
Рис.3. Результат выполнения скрипта

Константа может быть любого из существующих типов. Значение констант неизменно. Попытка изменения значения константы вызовет ошибку в программе.

В РНР есть несколько встроенных констант и поэтому возникает опасность совпадения имен встроенной и вашей константы, что вызовет предупреждения об ошибке. Чтобы узнать существует ли константа, можно использовать функцию defined().

Общий вид этой функции следующий:

  
Рис.4. Общий вид функции defined()

Она возвращает 1, если константа определена, и 0 - в противном случае. Приведем небольшой пример использования функции defined().

<?php

***//проверка на существование константы***

if (defined("CONSTANT"))

{

***// вывести сообщение, если константа существует***

echo "Константа определена";

}

?>

В примере приведена конструкция if, которая называется условной. Более подробно мы рассмотрим ее позднее.

Как мы уже говорили, в PHP заранее определен ряд констант. Они называются *предопределенными*. Их основное назначение - хранить информацию о системе. Например, PHP\_VERSION и PHP\_OS содержат, соответственно, версию РНР и название операционной системы, на которую установлен сервер.

Существуют так же несколько "волшебных" констант. Они могут менять свое значение в зависимости от их использования. Например, константы \_\_LINE\_\_ и \_\_FILE\_\_ содержат в себе, соответственно, номер строки и имя файла сценария. Обратите внимание, что эти константы можно писать как прописными буквами, так и заглавными.

**Операторы. Оператор присваивания**

***Оператор*** - конструкция языка, предназначенная для получения нового значения.

Первый оператор который мы изучим, будет ***оператор присваивания***, общий вид которого следующий:

http://it.kgsu.ru/PHP/images/ris7_1.jpg  
Рис.1. Общий вид оператора присваивания

Например:

**$a=14;**

**$ric="Care";**

**$b=2+$a;**

***К моменту выполнения оператора присваивания все переменные, входящие в выражение, стоящее справа от знака присваивания, должны быть определены.***

Выражение, как правило, содержит арифметические операции. Они вам давно известны. Это сложение (+), вычитание (-), умножение (\*), деление (/), остаток отделения (%).

Следующий пример иллюстрирует их использование:

**<?php**

**$a=12;**

**$b=4;**

**$c=$a+$b;**

**echo "12+4 = ";**

**echo $c;**

**echo "<br>";**

**$c=$a-$b;**

**echo "12-4 = ";**

**echo $c;**

**echo "<br>";**

**$c=$a\*$b;**

**echo "12\*4 = ";**

**echo $c;**

**echo "<br>";**

**$c=$a/$b;**

**echo "12/4 = ";**

**echo $c;**

**echo "<br>";**

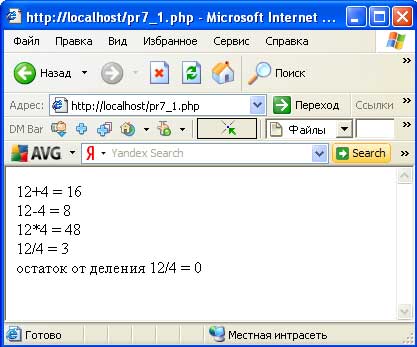
**$c=$a%$b;**

**echo "остаток от деления 12/4 = ";**

**echo $c;**

**?>**

Результат выполнения программы вы можете увидеть на рисунке 2.



**Операторы. Операторы отношения и логические операторы**

Следующая группа операторов - это ***операторы отношения***:

**$a == $b** - проверка на равенство;

**$a != $b** - проверка на неравенство;

**$a < $b** - проверка на "меньше";

**$a > $b** - проверка на "больше";

**$a <= $b** - проверка на "меньше либо равно";

**$a >= $b** - проверка на "больше либо равно";

**$a === $b** - проверка на идентичность, эквивалентность (**$a** и **$b** должны быть не только равны, но дополнительно они должны быть одинакового типа).

Дальше рассмотрим ***логические операторы***:

**$a && $b** или **$a and $b** - логическое "и";

**$a || $b** или **$a or $b** - логическое "или";

**$a XOR $b** - логическое "исключающее или";

**!$a** - логическое "не".

Таблицы значений логических операторов приведены ниже.

|  |  |  |
| --- | --- | --- |
| Таблица 1. **Логическое "и"** | | |
| **$а** | **$b** | **Результат** |
| **FALSE** | **FALSE** | **FALSE** |
| **FALSE** | **TRUE** | **FALSE** |
| **TRUE** | **FALSE** | **FALSE** |
| **TRUE** | **TRUE** | **TRUE** |

|  |  |  |
| --- | --- | --- |
| Таблица 2. **Логическое "или"** | | |
| **$а** | **$b** | **Результат** |
| **FALSE** | **FALSE** | **FALSE** |
| **FALSE** | **TRUE** | **TRUE** |
| **TRUE** | **FALSE** | **TRUE** |
| **TRUE** | **TRUE** | **TRUE** |

|  |  |  |
| --- | --- | --- |
| Таблица 3. **Исключающее "или"** | | |
| **$а** | **$b** | **Результат** |
| **FALSE** | **FALSE** | **FALSE** |
| **FALSE** | **TRUE** | **TRUE** |
| **TRUE** | **FALSE** | **TRUE** |
| **TRUE** | **TRUE** | **FALSE** |

|  |  |
| --- | --- |
| Таблица 4. **Логическое "не"** | |
| **$а** | **Результат** |
| **FALSE** | **TRUE** |
| **TRUE** | **FALSE** |

**Операторы. Инкремент и декремент**

В **PHP** есть такие операторы как ***инкремент*** и ***декремент***.

Инкремент: **i++;** аналогичен конструкции **i = i+1;**.

Декремент: **i--;** аналогичен конструкции **i = i-1;**.

Данные операторы являются унарными. Инкремент и декремент имеют ***префиксную*** (++i) и ***постфиксную*** (i++) формы. В чем их отличие разберемся на примере:

**<?php**

**$a = 1;**

**$b = 1;**

**$c = $a++ + 2; *// оператор справа***

***// сначала прибавили 2 к $a, а затем увеличили $a на 1***

**$d = ++$b + 2; *// оператор слева***

***// сначала увеличили $b на 1, а затем прибавили 2 к $b***

**echo "a = ";**

**echo $a; *// выводит 2***

**echo "<br>";**

**echo "b = ";**

**echo $b; *// выводит 2***

**echo "<br>";**

**echo "c = ";**

**echo $c; *// выводит 3***

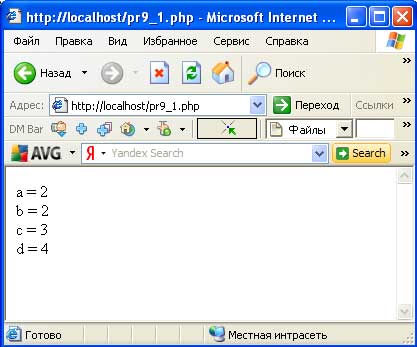
**echo "<br>";**

**echo "d = ";**

**echo $d; *// выводит 4***

**?>**

Результат выполнения программы вы можете увидеть на рисунке 1.

  
Рис.1. Префиксная и постфиксная форма инкремента и декремента

В первом случае оператор находится справа от операнда (**$а**). При этом сначала выполняется сложение, а затем уже увеличение операнда. Во втором случае все происходит наоборот: сначала выполняется увеличение операнда, а затем сложение. Этим и объясняется результат выполнения программы.

**Операторы. Стандартные математические функции**

Следующая группа операторов - это ***стандартные математические функции***. Основные математические функции перечислены в таблице 1.

|  |  |
| --- | --- |
| Таблица 1. **Основные математические функции** | |
| **Название функции и ее обозначение** | **Обозначение в языке PHP** |
| **sin x** | **sin($x)**, аргумент **$x** должен быть выражен в радианах |
| **cos x** | **cos($x)**, аргумент **$x** должен быть выражен в радианах |
| **tg x** | **tan($x)** |
| **arccos x** | acos($x) |
| **ex** | **exp($x)** |
| **logn x** | **log($x, $n)** |
| **log10 x** | **log10($x)** |
| **xn** | **pow($x, $n), $n** - степень (натуральное число) |
| Квадратный корень из **x** | **sqrt($x)** |
| **|x|** (модуль **x**) | **abs($x)** |
| Округление до целых | **round($x)** |
| Генерация случайного числа. Возвращает псевдослучайное целое в диапазоне от **X** до **Y** | **rand($x, $y)** |

Приведем пример применения стандартных математических функций:

**<?php**

**$a = 45;**

**$b = 2;**

**$c = 16;**

**$d=-5;**

**$e=12.6;**

**echo "sin (45) = ";**

**echo sin($a);**

**echo "<br>";**

**echo "exp (2) = ";**

**echo exp($b);**

**echo "<br>";**

**echo "log от 16 с основанием 2 = ";**

**echo log($c,$b);**

**echo "<br>";**

**echo "16^2 = ";**

**echo pow($c,$b);**

**echo "<br>";**

**echo "|-5| = ";**

**echo abs($d);**

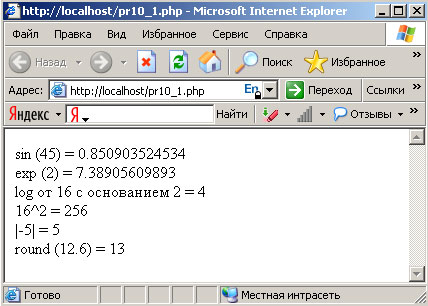
**echo "<br>";**

**echo "round (12.6) = ";**

**echo round($e);**

**?>**

Результат выполнения программы вы можете увидеть на рисунке 1.

  
Рис.1. Пример использования математических функций

**Операторы. Операция конкатенации**

Среди строковых операторов можно выделить один оператор - это ***оператор конкатенации (склеивание, сложение)*** - "." (точка). Этот оператор склеивает две строки.

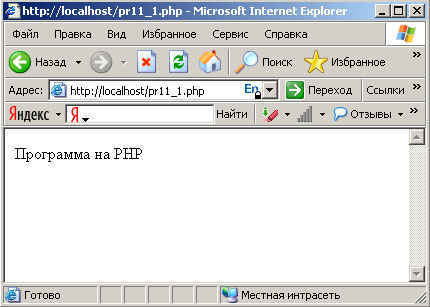
Приведем пример применения конкатенации:

**<?php**

**echo "Программа "."на "."PHP";**

**?>**

Результат выполнения программы вы можете увидеть на рисунке 1.

  
Рис.1. Пример использования конкатенации

**Операторы. Сокращенная запись операций**

Для некоторых операций из рассмотренных операторов существуют ***сокращенные записи***:

**$а += $b** - эквивалент **$а = $a + $b**;

**$а -= $b** - эквивалент **$а = $a - $b**;

**$а /= $b** - эквивалент **$а = $a / $b**;

**$а \*= $b** - эквивалент **$а = $a \* $b**;

**$а %= $b** - эквивалент **$а = $a % $b**;

**$а .= $b** - эквивалент **$а = $a . $b**.

Приведем пример применения сокращенных записей:

**<?php**

**$a=12;**

**$b=4;**

**$b=$a+$b;**

**echo "b = a+b";**

**echo "<br>";**

**echo "b = ";**

**echo $b;**

**echo "<br>";**

**$b=4;**

**$b+=$a;**

**echo "b += a";**

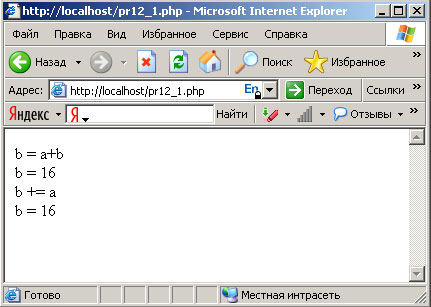
**echo "<br>";**

**echo "b = ";**

**echo $b;**

**?>**

Результат выполнения программы вы можете увидеть на рисунке 1.

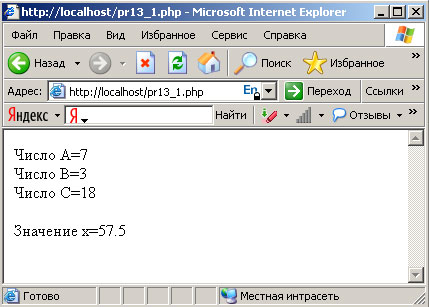
  
Рис.1. Пример использования сокращенной записи

Рассмотрим ***пример использования рассмотренных ранее конструкций***.

Создадим скрипт для решения следующей задачи: написать программу, которая для заданных значений **A, B, C** будет выводить результат выражения **x=A\*(B/6)+3\*C**.

Решением этой задачи будет следующий **php**-скрипт:

<?php

$A=7;

$B=3;

$C=18;

*// выводит значение переменной $A*

echo "Число A=".$A."<br>";

*// выводит значение переменной $B*

echo "Число B=".$B."<br>";

*// выводит значение переменной $C*

echo "Число C=".$C."<br><br>";

$x=$A\*($B/6)+3\*$C; *// вычисляет значение выражения*

echo "Значение x=".$x; *// выводит результат вычисления*

?>

    Данная программа выполнена при значениях $A=7, $B=3, $C=18.

    Задавать значения A, B, C в самом php-скрипте непрактично, так как необходимо будет каждый раз открывать файл с программой, если нужно будет проверить ее работу при других значениях. Чтобы этого избежать изменим содержимое файла следующим образом: создадим текстовые поля для каждого значения. В результате получим следующую программу:

<html>

<head>

<title>Решение задачи</title>

<meta charset="utf-8">

</head>

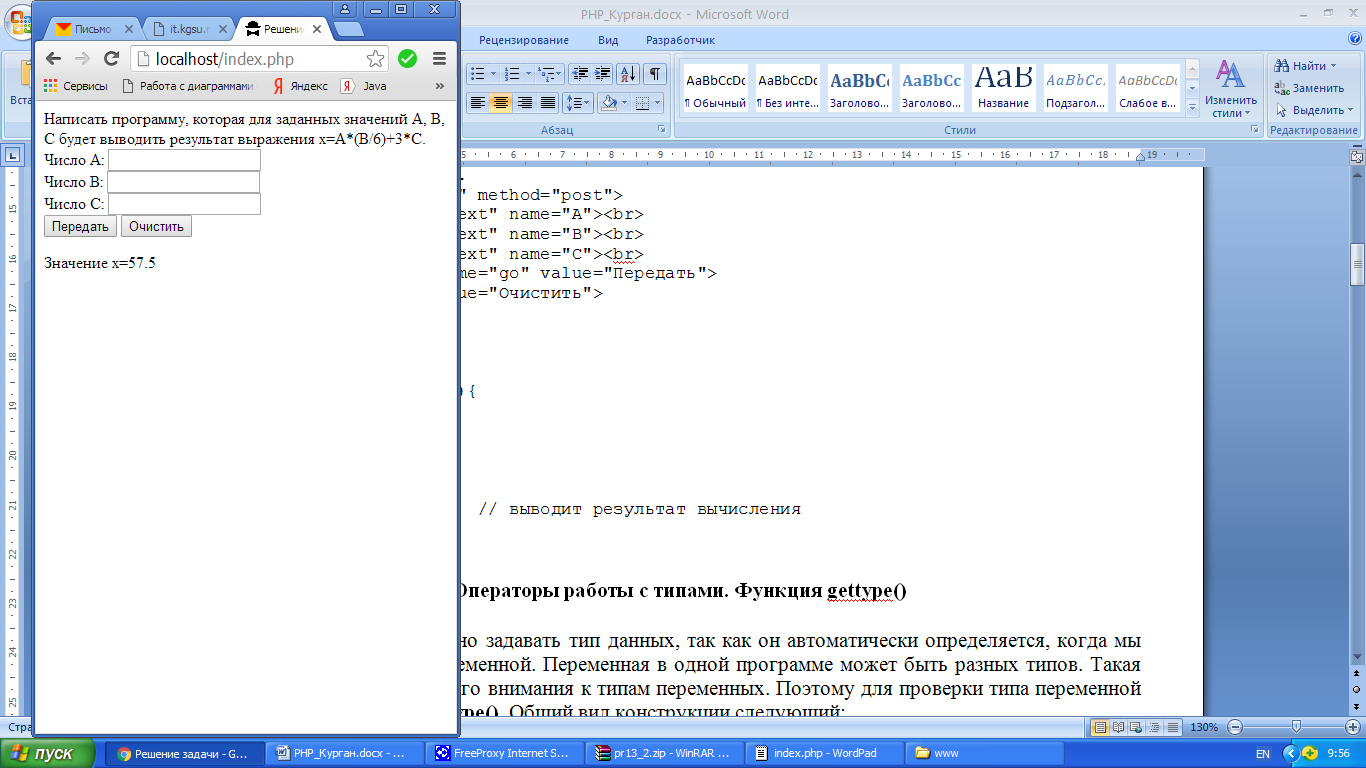
<body>

Написать программу, которая для заданных значений A, B, C будет выводить результат выражения x=A\*(B/6)+3\*C.

<form action="index.php" method="post">

Число A: <input type="text" name="A"><br>

Число B: <input type="text" name="B"><br>

Число C: <input type="text" name="C"><br>

<input type="submit" name="go" value="Передать">

<input type="reset" value="Очистить">

</form >

<?php

if (isset($\_POST['go'])){

$A=$\_POST['A'];

$B=$\_POST['B'];

$C=$\_POST['C'];

$x=$A\*($B/6)+3\*$C;

}

echo "Значение x=".$x; // выводит результат вычисления

?>

</body>

</html>

**Операторы работы с типами. Функция gettype()**

 В **PHP** не обязательно задавать тип данных, так как он автоматически определяется, когда мы присваиваем значение переменной. Переменная в одной программе может быть разных типов. Такая вольность требует большего внимания к типам переменных. Поэтому для проверки типа переменной применяется функция **gettype()**. Общий вид конструкции следующий:

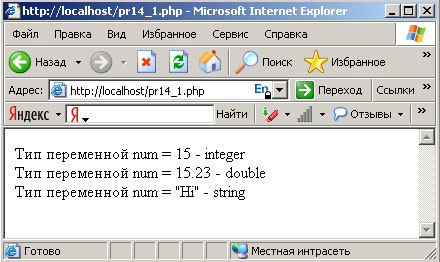


Данный оператор возвращает тип заданной переменной. Например:

<?php

$num = 15;

echo "Тип переменной num = 15 - ";

echo gettype($num); //выведет integer

echo "<br>";

$num = 15.23;

echo "Тип переменной num = 15.23 - ";

echo gettype($num); // выведет double

echo "<br>";

$num = "Hi";

echo "Тип переменной num = \"Hi\" - ";

echo gettype($num); // выведет string

?>

Если необходима проверка переменной на соответствие определенному типу данных, то применяются функцииis\_integer(), is\_double(), is\_string(), is\_array(), is\_object() и is\_bool().

Данные функции является функциями логического типа, то есть в случае соответствия типа переменной выводится 1, иначе - 0.

Приведем общий вид функции is\_integer():



Приведем пример использования указанных конструкций.

<?php

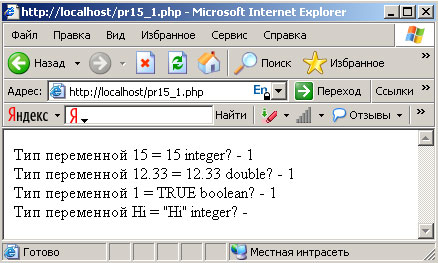
$test\_var = 15; *// переменная целого типа*

echo "Тип переменной $test\_var = 15 integer? - ";

echo is\_integer($test\_var); *// выводит 1*

echo "<br>";

$test\_var = 12.33; *// переменная вещественного типа*

echo "Тип переменной $test\_var = 12.33 double? - ";

echo is\_double ($test\_var); *// выводит 1*

echo "<br>";

$test\_var = TRUE; *// переменная логического типа*

echo "Тип переменной $test\_var = TRUE boolean? - ";

echo is\_bool($test\_var); *// выводит 1*

echo "<br>";

$test\_var = "Hi"; *// переменная стрового типа*

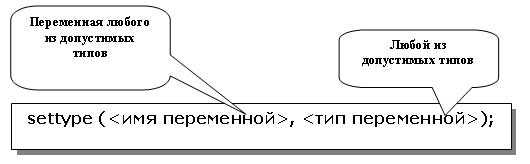
echo "Тип переменной $test\_var = \"Hi\" integer? - ";

echo is\_integer($test\_var); *// ничего не выведет*

?>

В РНР имеется возможность изменять тип значения переменной. В этом случае происходит преобразование значения переменной из одного типа в другой. Для этого используется функция settype().

    Общий вид этой функции следующий:



    Например:

<?php

$test\_var = 12.63; *// присваиваем значение 12.63*

echo gettype ($test\_var); *// выводит double*

echo "<br>";

echo $test\_var; *// выводит 12.63*

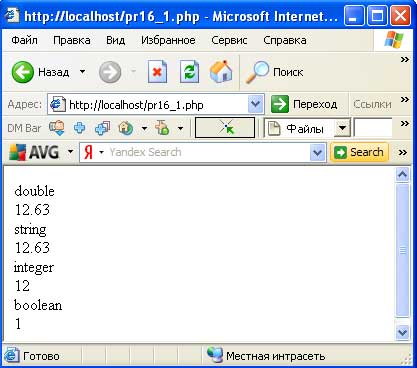
echo "<br>";

settype ($test\_var, string); *// устанавливаем тип string*

echo gettype ($test\_var); *// выводит string*

echo "<br>";

echo $test\_var; *// выводит "12.63"*

echo "<br>";

settype ($test\_var, integer); *// устанавливаем тип integer*

echo gettype ($test\_var); *// выводит integer*

echo "<br>";

echo $test\_var; *// выводит 12*

echo "<br>";

settype ($test\_var, boolean); *// устанавливаем тип boolean*

echo gettype ($test\_var); *// выводит boolean*

echo "<br>";

echo $test\_var; *// выводит 1*

echo "<br>";

?>

В начале мы присвоили переменной $test\_var значение 12.63, и автоматически ее тип стал Double, о чем свидетельствует вывод результата функции gettype() в окне браузера. Затем преобразовали тип переменной в String и опять вывели значение (теперь уже 12.63 является ни числом, а строкой). При преобразовании вInteger дробная часть отбрасывается, а не округляется до большего целого и остается 12. При изменении типа переменной на Boolean результат становится 1.

Нетрудно заметить, что преобразование типов данных осуществляется по определенным правилам. Приведем некоторые из них.

***Преобразование в тип*** **Boolean.**

1. Значения, преобразуемые в **FALSE**:
   * целое число 0;
   * дробное число 0.0;
   * пустая строка или строка "0".
2. Любые другие значения преобразуются в **TRUE**.

***Преобразование в тип*** **Integer.**

    Преобразование осуществляется таким образом:

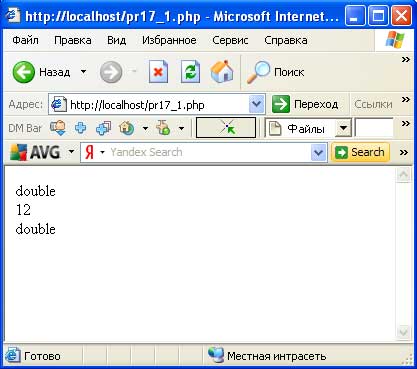
* значение **FALSE** преобразуется в 0, а **TRUE** - в 1;
* у вещественных чисел отбрасывается дробная часть;
* если строка не начинается с цифры, то она преобразуется в 0. Иначе будет указанное целое число.

***Преобразование в тип*** **String.**

* Значение **FALSE** преобразуется в пустую строку, а **TRUE** - в 1.
* Любые числа преобразуются в строку, содержащую цифры этих чисел.

***Приведение типов данных.***

    Приведение типов данных используется в тех случаях, когда изменение типа значения переменной не требуется.

<?php

$test\_var = 12.63;

echo gettype($test\_var); ***// выводит double***

echo "<br>";

echo (integer)$test\_var; ***// выводит 12***

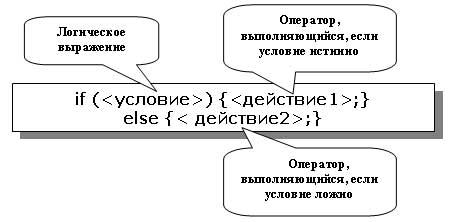
echo "<br>";

echo gettype($test\_var); ***// выводит double***

?>

**Условные операторы. Оператор if**

Оператор if можно встретить во многих языках программирования, и PHP не стал исключением. Общий вид оператора следующий:

  
Рис.1. Общий вид оператора if

Конструкция начинается со служебного слова if. Затем в круглых скобках записывается условие, которое необходимо проверить. Если это условие равно true (*истина*), то выполняется *действие1*. Иначе выполняется*действие2*. Например:

if ($d>0) $x+=10;

else $x-=5;

Здесь фигурные скобки опущены, так как *действие1* и *действие2* состоят из одного оператора. Если *действие1*или *действие2* состоит из нескольких операторов, то эти операторы заключаются в фигурные (операторные) скобки. Например:

if ($d>0)

{

$x+=10;

$d-=1;

}

else $x-=5;

Оператор if может употребляться как с конструкцией else, так и без нее. Например, если конструкция elseотсутствует, то такая форма записи называется *сокращенной*. Например:

if ($name =="Курган") $st = "столица Зауралья";

В этом случае, если значение переменной $name равно строке "Курган", то переменная $st получит в качестве значения строку "столица Зауралья". В противном случае значение никаких действий не выполнится, и значение переменной будет неопределенно (если эта переменная встречается в тексте программы первый раз), или не изменится (если эта переменная была определена ранее).

    Если конструкция else присутствует в условном операторе, то такая форма условного оператора называется*полной*.

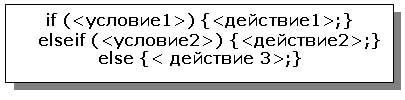
if ($k<>0) $n%=10;

else $n-=1;

В данном случае, если значение переменной $k не равно нулю, то значением переменной $n будет остаток от деления ее значения на 10; в противном случае (если значение переменной $k равно нулю), то новое значение переменной $n будет на единицу меньше предыдущего.

**Условные операторы. Оператор elseif**

Теперь рассмотрим оператор elseif. Он был введен для расширения возможностей условного оператора if. В общем случае эта конструкция выглядит так:

  
Рис.1. Общий вид оператора elseif

    Порядок ее выполнения приведен ниже:

if (условие1) действие1;

*/\* выполняется действие1, если условие1 истинно \*/*

elseif (условие2) действие2;

*/\* выполняется действие2, если условие1 ложно и условие2 истинно \*/*

else действие3;

*/\* выполняется действие3, если условие1 ложно и условие2 ложно \*/*

    Проиллюстрируем ее использование на следующем примере:

<?php

if ($a > $b) // $a больше $b?

{

*// если да, то выводим, и остальное пропускаем*

echo "а больше b";

}

*/\* если нет, то проверим, равно ли значение переменной $а значению переменной $b? \*/*

elseif ($a == $b)

{

*// если да, то выводим эту фразу, и остальное пропускаем*

echo "а равно b";

}

else

*/\* если ни одно из условий не выполнилось, то выводим данную фразу \*/*

{

echo "а меньше b";

}

?>

*Задание. Выполните приведенную условную конструкцию, задавая переменным $a и $b различные значения.*

    Приведем еще один пример. Необходимо по названию книги вывести фамилию ее автора:

<?php

$book\_name = "Самоучитель по PHP";

if ($book\_name == "Самоучитель по Perl")

{

echo "Автор: Петров"; *// выходные данные*

}

elseif ($book\_name == "Самоучитель no ASP")

{

echo "Автор: Иванов"; *// выходные данные*

}

elseif ($book\_name == "Самоучитель по PHP")

{

echo "Автор: Сидоров"; *// выходные данные*

}

?>

*Задание. Выполните приведенную условную конструкцию, задавая переменной $book\_name различные значения. Какой существенный недостаток вы нашли в этой программе?*

    При написании программ может потребоваться организовать проверку большого количества условий. В этом случае использование рассмотренных выше конструкций является не слишком эффективным. Для этих целей в язык программирования был включен оператор switch, к изучению которого мы сейчас переходим.

**Условные операторы. Оператор switch**

В общем случае эта конструкция выглядит так:

  
Рис.1. Общий вид оператора switch

Вся конструкция начинается со слова switch. Далее в круглых скобках следует выражение, которое, в отличие от выражения в операторе if, может быть не только логическим выражением (типа Boolean), но и целым числом (Integer), вещественным числом (Double) и строкой (String). От значения этого выражения зависит, какую группу операторов необходимо выполнить.

Затем следует блок, в котором находятся значения для сравнения и соответствующие им действия. Синтаксически это оформляется с помощью специального слова case и знака двоеточия.

Оператор break служит для того, чтобы выйти из конструкции switch в случае, когда значения выражения и указанной после служебного слова case константы, совпали.

После слова default следует действие, которое нужно выполнить, если ни одно из предыдущих действий не выполнилось, то есть не было совпадений ни с одним из значений.

Вернемся к примеру из [предыдущего шага](http://it.kgsu.ru/PHP/php019.html). Перепишем его, используя оператор switch.

<?php

$book\_name = "Самоучитель по PHP";

switch ($book\_name)

{

*// выводится, если $book\_name = "Самоучитель по Perl"*

case "Самоучитель по Perl": echo "Автор: Петров";

*// выход из конструкции switch*

break;

case "Самоучитель по ASP": echo "Автор: Иванов";

break;

case "Самоучитель по PHP": echo "Автор: Сидоров";

break;

*// сообщение выведется, если не было совпадений*

default: echo "Такой книги в наличии нет";

}

?>

*Задание. Выполните приведенную условную конструкцию, задавая переменной $book\_name различные значения. Исправлен ли в этом варианте существенный недостаток, найденный вами в предыдущем варианте этой программы?*

Надо отметить, что присутствие в конструкции switch слов default и break не является обязательным. Если опустить слово default, то в случае, когда совпадений не было, никаких действий просто не выполняется. Интереснее дело обстоит с оператором break. Давайте уберем этот оператор из приведенного примера. Текст программы будет выглядеть в этом случае так:

<?php

$book\_name = "Самоучитель по Perl";

switch ($book\_name)

{

case "Самоучитель по Perl": echo "Автор: Петров"."<br>";

case "Самоучитель по ASP": echo "Автор: Иванов"."<br>";

break;

case "Самоучитель по PHP": echo "Автор: Сидоров"."<br>";

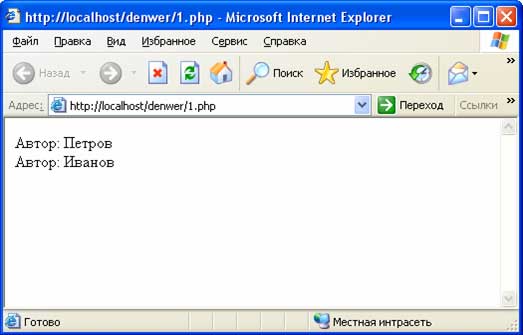
break;

default: echo "Такой книги в наличии нет"."<br>";

}

?>

Выполним ее. Сравните полученный вами результат с результатом, приведенный на рисунке 2.

  
Рис.2. Пример применения оператора switch

Прокомментируем полученный результат. Если совпали значение выражения с указанной после case константой и опущен оператор break, то после выполнения операторов, "помеченных" найденной константой, не произойдет выхода из оператора switch, а будут выполняться по порядку следующие операторы до тех пор, пока не будет найден оператор break.

Данная особенность в нашем случае привела к неправильной работе программы. Но в некоторых случаях это может оказаться полезным.

Изменим немного нашу программу. Пусть, например, Петров написал несколько книг. В этом случае задачу можно решить так, как показано на примере.

<?php

$book\_name = "Информатика в школе";

switch ($book\_name)

{

case "Самоучитель по Perl":

case "Информатика в школе":

case "Программирование в Internet": echo "Автор: Петров";

break;

case "Самоучитель по ASP": echo "Автор: Иванов";

break;

case "Самоучитель по PHP": echo "Автор: Сидоров";

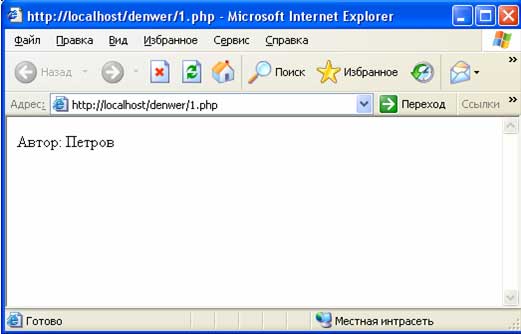
break;

default: echo "Такой книги в наличии нет";

}

?>

*Задание. Выполните приведенную условную конструкцию, задавая переменной $book\_name различные значения, и сравните результат с приведенным на рисунке 3.*

  
Рис.3. Пример применения оператора switch

В этом случае, если пользователь задал название любой из трех книг Петрова, выведется его фамилия, так как оператор break располагается только после названия последней книги этого автора.

В заключение отметим, что *значение*, следующие за case, в отличие от многих других языков программирования, может быть логическим выражением (тип Boolean), целым числом (Integer), вещественным числом (Double) или строкой (String). Например, в приведенных в этом разделе примерах, мы использовали строковые константы.

**Пример использования условных конструкций**

Приведем решение следующей задачи: написать программу, которая будет определять, лежит ли заданная точкаA(x1,y1) на прямой y=kx2/3 при заданном параметре k.

Решением этой задачи будет следующая программа:

<html>

<head>

<title>Решение задачи</title>

</head>

<body>

Написать программу, которая будет определять, лежит ли заданная точка A(x1,y1) на

прямой y=(k\*(x^2))/3 при заданном параметре k.

<form>

Координата x1 точки A: <input type="text" name="x1"><br>

Координата y1 точки A: <input type="text" name="y1"><br>

Коэффициент k прямой: <input type="text" name="k"><br>

<input type="submit" name="go" value="Передать">

<input type="reset" value="Очистить">

</form>

<?php

echo "Значение x1=".$x1."<br>";

echo "Значение y1=".$y1."<br>";

echo "Значение k =".$k."<br><br>";

$y=($k\*($x1\*$x1))/3;

echo "Значение y=".$y."<br>";

if ($y==$y1)

{

echo "Точка A принадлежит прямой";

}

else

{

echo "Точка A не принадлежит прямой";

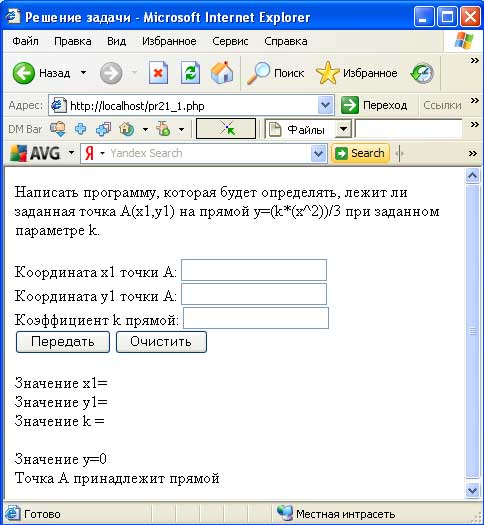
}

?>

</body>

</html>

В окне браузера форма будет выглядеть следующим образом (рисунок 1):

  
Рис.1. Форма в окне браузера

Когда пользователь нажмет кнопку *"Передать"*, браузер передаст сценарию следующие параметры:

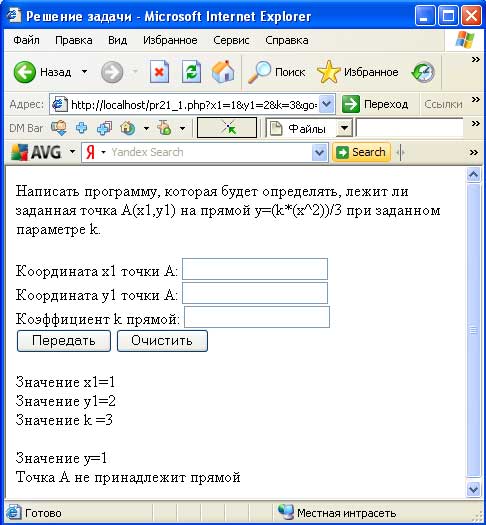
x1 - значение первого текстового поля, координата x точки A;

y1 - значение второго текстового поля, координата y точки A;

k - значение третьего текстового поля, коэффициент прямой.

Затем вычисляется значение выражения y. Если это значение совпадает с y1, то выводим на экран *"Точка A принадлежит прямой"*, в противном случае - *"Точка A не принадлежит прямой"*.

Введем конкретные значения: x1=1, y1=2, k=3 и нажмем кнопку *"Передать"*. Результат работы программы представлен на рисунке 2.

  
Рис.2. Результат работы программы

**Циклы (общие сведения)**

Без операторов цикла вообще трудно себе представить программирование. Например, нужно вывести на экран целые числа от 1 до 100. Один из способов решения этой задачи заключается в написании сто раз подряд конструкции echo. Ясно, что такая программа будет очень громоздкой. Чтобы эффективно решать подобные задачи как раз применяют операторы цикла.

Дадим определение цикла. *Многократно повторяющаяся последовательность операторов называется циклом, а программа, содержащая цикл, - циклической*.

Ясно, что практически все языки программирования содержат конструкции, позволяющие достаточно просто и эффективно организовывать циклы. Есть подобные конструкции в языке программирования PHP. Перейдем к их изучению.

**Циклы. Цикл for**

Эта конструкция цикла используется тогда, когда заранее известно количество повторений тела цикла.

Синтаксис цикла for следующий:

  
Рис.1. Общий вид цикла for

Вся конструкция начинается со служебного слова for. Далее в круглых скобках следуют через точку с запятой три выражения, после чего записывается выполняемое действие. Действие еще называют *телом цикла*.

Чтобы понять смысл этих выражений, рассмотрим решение примера по выводу целых чисел от 1 до 100.

<?php

for ($i = 1; $i <= 100; $i++)

{

echo $i;

echo "<br>";

}

?>

*Выражение1* задает начальные значения и определяется один раз перед входом в цикл: в данном случае переменной $i присваивается значение 1. *Выражение2* определяет условие нахождения в цикле. Другими словами, до тех пор, пока оно равно true (является истинным), программа будет продолжать выполнять тело цикла, т.е. в нашем примере, вывод значений переменной. И, наконец, *выражение3* представляет собой действие, которое нужно выполнить по завершению очередного выполнения тела цикла. Здесь это увеличение переменной $i на единицу.

Рассмотрим ход выполнения этой программы. Сначала значением переменной $i становится равным 1. Затем производится операция сравнения, то есть вычисляется *выражение2*. Если результат этого выражения истинен, то выполняется вывод сообщения (то есть выполняется тело цикла) и затем выполняется *выражение3*(переменная $iувеличивается на 1), иначе осуществляется выход из цикла. Следующая итерация начинается уже с вычисления*выражения2*.

Надо сказать, что разработчики РНР сделали цикл for настолько универсальным, что в нем можно уместить целые программы.

Особенность выражений оператора for в том, что они могут состоять из нескольких подвыражений. Например:

for ($i=1, $j=9; $i<=9, $j>=1; $i++, $j--);

Заметим, что подвыражения разделяются между собой запятыми. Пояснений, наверное, требует только*выражение2*, так как в других случаях здесь просто перечисляются несколько действий. В *выражении2*, если хотя бы одно из подвыражений становится ложным, то произойдет выход из цикла. То есть в *выражении2* подвыражения связаны между собой логической операцией "и".

Пожалуй, единственное, о чем мы еще не говорили, так это о теле цикла. В данном случае оно, как и у оператораif, может состоять из одного или нескольких операторов. В первом случае тело цикла можно не заключать в фигурные скобки (однако, их использование все же желательно), а во втором они необходимы.

**Циклы. Цикл while**

Следующий тип циклов, который мы рассмотрим, это цикл **while**. Общий вид этого оператора следующий:

  
Рис.1. Общий вид оператора **while**

Конструкция начинается со слова **while**, после которого следует выражение, имеющее такой же смысл, как и***выражение2*** у оператора **for**, то есть пока оно истинно, будет выполняться тело цикла.

Выражение оператора **while** не может состоять из подвыражений. Например, следующая конструкция вызовет ошибку.

**while ($i <= 9, $j >= 2) *//Ошибка!***

Для того, чтобы задать сразу несколько условий, применяются логические операторы.

**while ($i <= 9 && $j >= 2) *//Оператор выполнится***

Ход работы оператора **while** следующий. Сначала проверяется условие. Если это условие истинно, то выполняется тело цикла, иначе осуществляется выход из цикла. Приведем пример:

**<?php**

**$i = 1;**

**while ($i <= 100)**

**{**

**echo $i;**

**$i++;**

**}**

**?>**

Эта программа решает задачу вывода чисел от 1 до 100, только уже с применением цикла **while**.

***Задание.****Сравните решения этой задачи с циклом****for****и****while****. Объясните замеченные различия. Какой, по вашему мнению, цикл является более универсальным?*

**Циклы. Цикл do...while**

Рассмотрим последний вид цикла. Конструкция do...while используется в тех случаях, когда тело цикла должно выполниться хотя бы один раз. При этом условие выхода проверяется после выполнения тела цикла. Синтаксис цикла do...while следующий:

  
Рис.1. Общий вид оператора do...while

Конструкция начинается со слова do, за которым следует тело цикла. Затем записывается слово while с логическим выражением в круглых скобках. Приведем текст ранее рассмотренной задачи с использованием данной конструкции цикла:

<?php

$i = 1;

do

{

echo $i;

$i++;

}

while ($i <= 100);

?>

В данном примере сначала выполнится тело цикла, а затем будет проверяться условие. Если оно окажется ложным (равно false), то произойдет выход из цикла.

Когда используют конструкцию do...while, обычно допускают следующую ошибку. Многим кажется, что слова doи while могут выступать в роли фигурных скобок, то есть выделять блок операторов. Это совершенно не так! Приведем пример:

<?php

do

echo 1;

echo 2; *// эта строка вызовет ошибку*

while (0);

?>

Нужно обязательно помещать блок команд в фигурные скобки. Правильная запись выглядит так:

<?php

do

{

echo 1;

echo 2; *// эта программа будет выполнена*

}

while (0);

?>

**Безусловные операторы. Оператор break**

Безусловные операторы в основном предназначены для работы с циклами. В некоторых случаях они существенно упрощают алгоритм программы и делают код более понятным. Используйте их с особой осторожностью, так как в блоках программ, содержащих безусловные операторы, наиболее часто встречаются ошибки.

Оператор break

Первый безусловный оператор - break. Применение оператора break не ограничивается конструкцией switch. Он очень часто применяются в циклах. Например, мы хотим выяснить, есть ли книги в магазине объемом более 400 страниц, причем нас не интересует их количество. Предположим, что у нас имеется список этих книг с соответствующей информацией. Начиная с первой книги, мы будем сравнивать количество страниц с числом 400. Как только мы встретили книгу, удовлетворяющую нашему условию, надо вывести соответствующее сообщение и выйти из цикла. Именно эту функцию и выполняет оператор break.

На практике очень часто применяют *вложенные циклы*. Вложенным называется цикл, который располагается в теле другого цикла. Например:

<?php

for ($i=1;$i<=10;$i++)

{for ($j=1;$j<=10;$j++)

{$sum++;

if (($j+($i-1)\*10) == 55) break 2;

}

}

echo $sum;

?>

Отличительной чертой break в PHP является его параметр, указывающий количество циклов, из которых нужно выйти. По умолчанию он равен 1, что означает выход из текущего цикла. В нашем примере мы использовали параметр, который равен 2. В результате выход произошел на 55 итерации (это количество считает переменная $sum). Так как наш параметр равен 2, то вычисления завершили оба цикла.

*Задание. Попробуйте убрать этот параметр и объясните, почему получается именно такой результат.*

**Безусловные операторы. Оператор continue**

Следующий безусловный оператор - continue. Оператор continue так же, как и break, применяется в циклах. Его выполнение приводит к незамедлительному переходу к следующей итерации.

<?php

for ($x=-5; $x<=5; $x++)

{

if (!$x) continue;

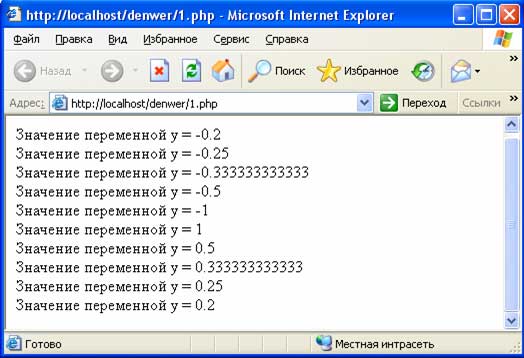
$y=1/$x;

echo "Значение переменной у = ".$y."<br>";

}

?>

Результат выполнения программы вы можете увидеть на рисунке 1.

  
Рис.1. Пример применения оператора continue

Здесь мы выводим значения переменной $у в зависимости от $х. В случае, когда переменная $x будет равна 0, в окне браузера должно быть выведено сообщение об ошибке, так как на 0 делить нельзя. Чтобы избежать такой ситуации, можно применить оператор if, а можно написать просто continue, и проблема будет решена. В этом случае (при $x равном 0) будет осуществлен переход к следующей итерации цикла.

*Задание. К какому оператору будет осуществлен переход после выполнения оператора continue?*

**Безусловные операторы. Оператор exit**

Последний безусловный оператор - exit. Оператор exit применяется, если нужно досрочно выйти из приложения.

Например, этот оператор часто применяется, когда произошла ошибка в программе и дальнейшее выполнение скрипта не имеет смысла:

<?php

if ($error == 1)

{echo "Произошла критическая ошибка!";

exit;

}

?>

**Пример использования цикла**

Приведем небольшой пример, иллюстрирующий использование циклических конструкций. Составить программу, которая будет выводить на экран квадраты чисел от **A** до **B** (**A** и **B** заданы), причем **B>=A**.

Вот текст программы, решающей данную задачу.

**<html>**

**<head>**

**<title>Решение задачи</title>**

**</head>**

**<body>**

**Составить программу, которая будет выводить на экран квадраты чисел от**

**A до B (A и B заданы), причем B >= A.**

**<form>**

**Число A: <input type="text" name="A"><br>**

**Число B: <input type="text" name="B"><br>**

**<input type="submit" name="go" value="Передать">**

**<input type="reset" value="Очистить">**

**</form>**

**<?php**

**echo " Число A=".$A."<br>";**

**echo " Число B=".$B."<br><br>";**

**for ($i=$A;$i<=$B;$i++)**

**{**

**$c=$i\*$i;**

**echo $i."²= ".$c."<br>";**

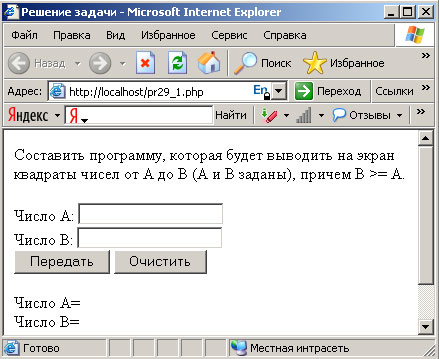
**}**

**?>**

**</body>**

**</html>**

В окне браузера форма будет выглядеть следующим образом (см. рис. 1):

  
Рис.1. Форма в окне браузера

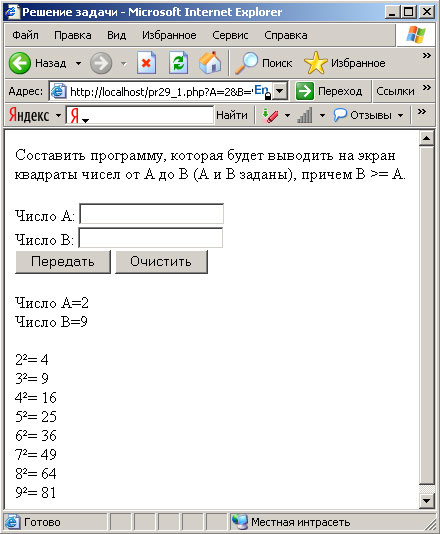
Когда пользователь нажмет кнопку ***"Передать"***, браузер передаст сценарию следующие параметры:

**A** - значение первого текстового поля, число **A**;

**B** - значение второго текстового поля, число **B**.

Затем начинается работа цикла. Его начальное значение будет равно введенному числу **$A**. Работа цикла закончится, когда переменная **$i** примет значение большее, чем **$B**. В теле цикла текущее значение переменной **$i**возводится в квадрат и затем полученное значение выводится на экран. После выполнения тела цикла **$i**увеличивается на 1.

Ведем конкретные значения: **A=2, B=9** и нажмем кнопку ***"Передать"***. Результат работы программы представлен на рисунке 2.

  
Рис.2. Результат работы программы

**Операторы require и include (общие сведения)**

Обычно крупное приложение не содержат в одном файле, а разбивают на несколько частей. Например, часто используемые константы, переменные и функции лучше хранить в отдельном файле и использовать его содержимое по необходимости. Именно эту функцию исполняют операторы require и include.

**Оператор require**

Рассмотрим оператор require. Его синтаксис очень простой:

http://it.kgsu.ru/PHP/images/ris31_1.jpg  
Рис.1. Синтаксис оператора require

Результатом его выполнения будет вставка текста из указанного файла при запуске программы (не при выполнении!). Этот файл может содержать как PHP-код, так и просто текст. Пример:

Файл global.php:

<?php

$a ="Переменная a находится в файле global.php";

?>

Файл prog.php:

<?php

require "global.php";

echo $a;

?>

В результате выполнения файла prog.php на экран будет выведена фраза: *Переменная* a *находится в файле*global.php.

**Оператор include**

Оператор include выполняет такие же действия, как и require, но с одним лишь отличием - он вставляет текст указанного файла во время выполнения программы.

<?php

for ($i=1; $i<=5; i++)

{

include "file".$i.".php";

}

?>

В данном примере вставляется содержимое пяти файлов. Заметьте, что использование оператора require привело бы к возникновению ошибки.

**Функции (общие сведения)**

Раньше исходный текст программы представлял собой машинный код, то есть последовательность нулей и единиц. Но человек постоянно пытался научить машину "говорить" с ним на одном языке. В результате непонятные для нас нолики и единички стали превращаться в команды, выполняющие одно действие. Но на этом развитие языков программирования не закончилось, так как с увеличением сложности решаемых задач количество команд возрастало в геометрической прогрессии. Для решения этой проблемы программисты создали структуру, с помощью которой можно было выделить блок операторов и придать ему определенное имя. Этот блок стали называть *подпрограммой* или *процедурой*. Это нововведение позволило резко сократить код программ и сделало его более понятным. Именно с этого момента начинается эпоха процедурного программирования. Дальнейшее развитие привело к тому, что у процедур появились входные параметры, которые назвали *аргументами*. И, наконец, появились функции, отличающиеся от процедур тем, что они могли не просто выполнять определённые действия, но и возвращать значения.

Тем не менее, в РНР нет понятия процедуры. Вне зависимости, есть ли возвращаемое значение или его нет, мы имеем дело с функцией.

В РНР есть множество встроенных функций, которые решают широкий круг стандартных задач. Но на практике даже такого большого количества функций бывает мало. Поэтому в РНР, как и во многих других языках программирования, имеется возможность создавать функции самостоятельно непосредственно в коде программы. Их часто называют *пользовательскими функциями*. Рассмотрим, как можно определить и использовать функцию.

**Определение функции**

Общий вид пользовательской функции следующий:

  
Рис.1. Общий вид функции

Описание пользовательской функции начинается со слова **function**. Затем следует ее ***имя*** и в круглых скобках через запятую указываются входные параметры (***аргументы***). Далее в фигурных скобках следует ***тело функции***, в котором указываются операторы, которые необходимо выполнить. Например:

**<?php**

**function error\_msg ($err\_str)**

**{**

**echo "<b>Ошибка!<br>Причина: " . $ err\_str . "</b>";**

**}**

**error\_msg("вы ввели отрицательное число");**

**?>**

В данном случае имя функции - **error\_msg**, а аргумент - **$err\_str**.

Разберем поэтапно ход работы такой программы. При вызове функции **error\_msg(*"вы ввели отрицательное число"*)** строка ***"вы ввели отрицательное число"*** записывается в переменную **$err\_str** и выводится в теле функции.

Заметьте, что описание функции мы поместили прежде, чем вызвали ее. На самом деле, начиная с четвертой версии **РНР**, описание может располагаться в любом месте программы.

Мы уже встречали среди встроенных функций те, которые возвращают значения. При создании пользовательских функций тоже имеется такая возможность.

**<?php**

**function mnog($num)**

**{**

**return $num\*$num;**

**}**

**echo mnog(2);**

**?>**

Эта функция подсчитывает квадрат числа, который передается в качестве параметра. Возвращение результата происходит посредством оператора **return**, который находится в теле функции. Все, что записывается после него, функция передает программе.

Прежде чем начинать писать свои функции, убедитесь в том, что среди встроенных функций нет функции, подобной вашей. Встроенные функции обычно работают куда быстрее и надежнее, чем пользовательские.

**Область видимости переменных**

При активной работе с пользовательскими функциями вы обязательно столкнетесь с проблемой области видимости переменных. Что это такое, лучше пояснить на примере:

<?php

function inc()

{

$num++;

}

$num = 1;

inc(); *// вызываем функцию*

echo $num; *// выведет 1*

?>

Результатом выполнения этой программы будет вывод числа 1. Другими словами, расположенные в теле функции и в основной программе переменные совершенно не связаны друг с другом, хотя имеют одинаковые имена. В этом случае говорят, что переменная, находящаяся внутри тела функции, имеет *локальную область видимости*, а переменная, которая располагается в основной программе, - *глобальную*. Локальные переменные объявляются внутри тела функции и недоступны извне. Глобальные переменные могут объявляться как в основной программе, так и в теле функции, но делается это с помощью специального оператора global. Например:

<?php

function inc()

{

global $num;

$num++;

}

$num = 1;

inc(); *// вызываем функцию*

echo $num; *// выведет 2*

?>

В этом случае выводится число 2, так как переменная $num в теле функции имеет глобальную область видимости.

**Время жизни переменных**

Такое понятие, как время жизни переменных, тоже очень распространено в программировании. Например, время жизни глобальных переменных начинается с того момента, как их объявили, и заканчивается в двух случаях. Либо их уничтожили непосредственно в программе, например с помощью функции **unset()**, либо завершилась работа сценария.

Общий вид функции **unset()** следующий:

  
Рис.1. Общий вид функции **unset()**

Функция **unset()** удаляет перечисленные переменные. Приведем пример использования этой функции:

**<?php**

**$global\_var = 1;**

**unset($global\_var);**

**echo $global\_var;**

**?>**

В результате программа выводит пустую строку, так как функция **unset()** уничтожила переменную **$global\_var**.

Поведение **unset()** внутри пользовательской функции может различаться в зависимости от того, какой тип имеет переменная, которую необходимо удалить.

Если переменная, объявленная глобальной, удаляется внутри пользовательской функции, то работать с ней в этой функции мы уже не сможем. Она станет доступна тогда и только тогда, когда будет осуществлен выход из этой функции. В этом случае глобальная переменная сохранит то же значение, что и до вызова **unset()**.

Иначе дело обстоит с локальными переменными, так как время их жизни напрямую зависит от продолжительности выполнения пользовательской функции. Однако встречаются такие ситуации, когда нужно сохранять значения локальных переменных. Для этого применяют специальный оператор **static**.

**<?php**

**function print\_msg($msg)**

**{**

**static $num = 1;**

**echo $num .") ". $msg . "<br>";**

**$num++;**

**}**

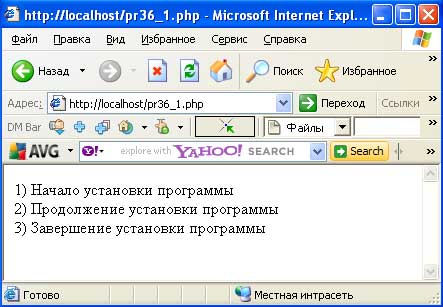
**print\_msg("Начало установки программы");**

**print\_msg("Продолжение установки программы");**

**print\_msg("Завершение установки программы");**

**?>**

Результат выполнения программы вы можете увидеть на рисунке 2.

  
Рис.2. Результат работы скрипта

Из примера видно, что переменная **$num** определяется один раз при первом вызове и не уничтожается после завершения работы функции, так как во втором вызове мы имеем дело со значением предыдущего вызова.

Пользовательская функция должна быть небольшой по объему. Обычно программисты говорят, что функция должна умещаться на экране монитора (приблизительно 20-30 строк). Это очень удобно при написании программы.

**Рекурсия**

В программировании имеется понятие "рекурсия".

Например, попробуем написать функцию с помощью обычного цикла for, возвращающую факториал числа. Он вычисляется следующим образом: n!=1\*2\*...\*(n-1)\*n. Причем 0! = 1 и 1! = 1, а факториала отрицательных чисел не бывает. Напишем функцию нахождения факториала числа без рекурсии.

<?php

function factorial ($num)

{

if ($num < 0)

{

return 0;

}

if ($num == 0)

{

return 1;

}

for ($i = 1, $sum = 1; $i <= $num; $i++)

{

$sum = $sum \* $i;

}

return $sum;

}

echo factorial(3); *//выведет 6*

?>

В этом случае выводится число 6. Задача решена, как говорится, "в лоб". В случае передачи отрицательных значений функция будет возвращать 0. Это реализуется с помощью первого оператора if. Если передано число 0, то возвращаем единицу. При любых других значениях факториал вычисляем в цикле for.

При решении задач с помощью рекурсии требуются рассуждения иного рода. Например, факториал числа можно вычислить следующим образом:

n! = 1, если n = 0 и n = 1,

n! = n\*(n-1)!, если n>0.

Заметьте, что второе утверждение содержит в себе знак факториала. Именно в этом и состоит основной смысл рекурсии. При решении определенной задачи мы используем то, что нам нужно найти. В программировании*функцию называют рекурсивной, если в ее описании присутствует обращение к самой себе*.

Итак, вернемся к задаче о факториале числа. При решении задач с помощью рекурсии всегда надо иметь два типа утверждений: *базисное* и *рекурсивное*. В нашем случае базисным утверждением является n! = 1 *если* n = 0 *или* n = 1. Эти значения называются *точками останова*. Работая с рекурсивным утверждением, мы должны перейти к базисному и получить результат. Теперь напишем функцию нахождения факториала числа с использованием рекурсии.

<?php

function factorial ($num)

{

if ($num < 0)

{

return 0;

}

if ($num == 0)

{

return 1;

}

return $num\*factorial($num-1); *// рекурсивный вывод*

}

echo factorial(3); *// выведет 6*

?>

В этом случае первые два условия взяты из предыдущего примера. Но теперь вместо цикла for мы записываем рекурсивное выражение, в котором функция вызывает сама себя. В нашей программе это происходит до тех пор, пока в качестве входного параметра не будет число 0, при котором функция возвратит 1. Причем это значение возвратится не в основную программу, а в тело предыдущей функции. Эта функция, в свою очередь, возвратит свое значение в тело следующей функции и т.д., пока нужное значение не возвратится в основную программу.

**Понятие массива**

Представьте себе, что приложение работает с данными, которые содержат в себе тысячи записей. Такое очень часто встречается на практике: например, работа с базой данных телефонных номеров. До этого мы разбирали переменные, которые позволяли хранить единственное значение. Естественно, с их помощью можно работать с любым количеством данных, но только сложно себе представить человека, способного разобраться в коде, содержащем тысячи переменных. В такой программе очень велика вероятность появления ошибки, а ее нахождение будет подобно поиску иголки в стоге сена. Именно по этим и другим причинам почти во всех языках программирования существуют переменные способные хранить в себе множество значений. Одной из них является ***массив***.

Мы уже говорили, что существует такой тип данных, как **array**. Переменные этого типа называются ***массивами***. При работе с массивами почти не существует жестких рамок, что позволяет программисту проявить свою фантазию.

Массив представляет собой набор элементов, каждый из которых имеет значение и ключ (индекс). ***Значение*** - это данные, которые хранит элемент массива, а по ***ключу*** мы можем обратиться к нему. Для лучшего понимания, о чем идет речь, представьте себе шкафчики для одежды, которые обычно находятся в спортивных раздевалках. Чтобы человек не забыл, где он оставил свою одежду, на дверцу наносят порядковый номер. Так вот, шкафчик - это элемент массива, его содержимое (одежда) - значение, а порядковый номер - ключ. Эта простая аналогия очень часто позволяет понять суть массивов.

Синтаксис массивов почти не отличается от других языков программирования.

Например:

**$mas[2];**

В этом примере мы обращаемся к элементу массива с именем **$mas**, который имеет ключ в виде числа 2. Надо отметить, что правила выбора имени массива такие же, как и у обычных переменных. Ключ записывается в квадратных скобках.

Особенность массивов **РНР** заключается в том, что элементы одного массива могут быть хранить данные разного типа. Возвращаясь к нашим шкафчикам, нужно сказать, что в них, например, можно хранить не только одежду, но и спортинвентарь (мячи, ракетки и тому подобное).

Для человека привычнее все элементы массива нумеровать, то есть в качестве ключа использовать число. Но иногда встречаются такие ситуации, когда удобно вместо числа использовать строку. Например, на каждом шкафчике писать имя его владельца. В качестве ключа в **РНР** может быть либо целое число (**Integer**), либо строка (**String**).

**Инициализация массивов**

Для инициализации или создания массивов в РНР предлагается два способа:

*присвоение значений* и

[*с помощью функции* array()](http://it.kgsu.ru/PHP/php039.html#array).

*Присвоение значений.* Для того чтобы создать массив, можно просто присвоить значение его элементу подобно тому, как мы делали это с обычными переменными:

$closets[3] = "Майка";

В результате выполнения этой строки если массив $closets (*шкафчики*) еще не существует, то он будет создан. Его первый и пока единственный элемент будет содержать строку *Майка*. Ключом в данном случае будет число 3. Ключ может и отсутствовать. Пример:

$closets[] = "Майка";

Здесь в квадратных скобках мы ничего не поставили. В этом случае если такого массива еще не существует, то по умолчанию ключом первого элемента будет число 0. Эта запись будет эквивалентна следующей:

$closets[0] = "Майка";

Если в качестве ключа используется строка, тогда инициализация массива будет выглядеть следующим образом:

$closets["Петров"] = "Майка";

В этом случае мы создали массив с одним элементом, ключом которого является строка *"Петров"*.

В PHP существует особенность, связанная с использованием пустого ключа ([]). Например:

$closets[] = "Майка";

$closets[] = "Кроссовки";

$closets[] = "Шорты";

Первой строкой создается массив с одним элементом, ключом которого является число 0. При выполнении следующей строки РНР добавит еще один элемент, ключ которого на единицу больше, то есть 1. Другими словами все это эквивалентно следующему:

$closets[0] = "Майка";

$closets[1] = "Кроссовки";

$closets[2] = "Шорты";

Общий механизм подсчета очередного ключа при использовании пустых квадратных скобок нужно отдельно пояснить. Допустим, массив уже существует, тогда осуществляется поиск максимального числового ключа и именно к нему прибавляется единица. Пример:

$closets[10] = "Майка";

$closets[1] = "Кроссовки";

$closets["Петров"] = "Шорты";

$closets[] = "Брюки"; *// эквивалентно $closets[11] = "Брюки"*

В этом случае четвертым элементом будет строка *"Брюки"*, а ключ будет равен 11, так как максимальное значение ключа до этого было равно 10. Если числовые ключи отсутствуют, то по умолчанию он будет равен 0, Пример:

$closets["Петров"] = "Шорты";

$closets[] = "Брюки"; *// эквивалентно $closets[0] = "Брюки";*

Исключение из этого правила может возникнуть при использовании функции unset(). Как эта функция применяется к массивам, мы рассмотрим позже.

*Функция* array(). Другим способом инициализации массивов является функция array().

Общий вид этой функции следующий:

  
Рис.1. Общий вид функции array()

Например:

$closets = array(1 => "Майка", 2 => "Кроссовки", 3 => "Шорты");

Функция array() в качестве входных параметров принимает пары ключ-значение и возвращает требуемый массив. Ключ и значение разделяются оператором =>. Пары разделяются запятыми. В приведенном примере создается массив, состоящий из трех элементов с ключами 1, 2 и 3 и значениями соответственно *Майка, Кроссовки, Шорты*.

Ключи можно не указывать, тогда их значение будет определяться по умолчанию. Например:

$closets = array("Майка", "Кроссовки", "Шорты");

В этом случае индексация будет начинаться с 0, поэтому рассматриваемая строчка будет эквивалентна этой:

$closets = array(0 => "Майка", 1 => "Кроссовки", 2 => "Шорты");

Правила определения ключей по умолчанию здесь такие же, что и рассмотренные раньше. И не стоит забывать, что в качестве ключей могут использоваться и строки:

$closets = array("Петров" => "Майка",

"Иванов" => "Кроссовки",

"Сидоров" => "Шорты");

**Вывод массивов**

После того, как вы освоили инициализацию массивов, нужно научиться их использовать. Для начала попробуем просто вывести массив на экран:

<?php

$closets = array(0 => "Майка", 1 => "Кроссовки", 2 => "Шорты");

echo $closets;

?>

В результате в окне браузера выведется слово Array!. Ошибки в выполнении программы здесь нет, так как она просто выводит слово, которое указывает, что данная переменная является массивом. Просто в этом случае программист не видит разницы между переменной скалярного и смешанного типа (к последнему и относятся массивы).

Наверное, самым удобным способом, которым можно вывести массив, является функция print\_r().

http://it.kgsu.ru/PHP/images/ris40_1.jpg  
Рис.1. Общий вид функции print\_r()

Эта функция может вывести значения массива с ключами. Например:

<pre>

<?php

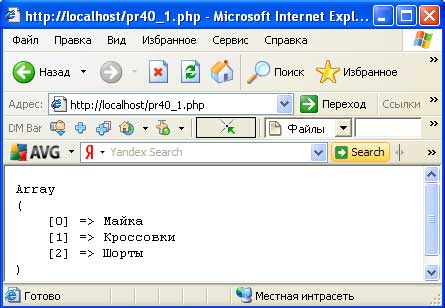
$closets = array(0 => "Майка", 1 => "Кроссовки", 2 => "Шорты");

print\_r($closets);

?>

</pre>

    Результат выполнения программы вы можете увидеть на рисунке 2.

  
Рис.2. Результат работы скрипта

На самом деле возможности функции print\_r() куда шире, чем просто вывод массивов, но здесь мы не будем приводить ее полное описание.

**Обход массивов. Функция count()**

На этом шаге мы рассмотрим *назначение и использование функции*count().

Функция print\_r() позволяет посмотреть на весь массив целиком, но не может выделить отдельные его части. Представьте себе задачу, когда требуется вывести из массива элементы, удовлетворяющие определенному условию. В этом случае функция print\_r() не сможет нам помочь. Для решения подобных задач применяют операторы цикла. Но и здесь нас ожидают "подводные камни". Сначала разберем самый простой случай, когда массив в качестве ключей имеет последовательный ряд чисел. Воспользуемся циклом for. Единственное, что нам надо выяснить, - это количество элементов в массиве. На практике очень часто для решения данной проблемы применяют функциюcount(). Общий вид функции count() следующий:

http://it.kgsu.ru/PHP/images/ris41_1.jpg  
Рис.1. Общий вид функции count()

В качестве входного параметра для данной функции является массив, а возвращает она количество элементов в нем.

Приведем небольшой пример, иллюстрирующий ее использование:

<?php

*// инициализация массива*

$closets = array(0=>"Майка", 1=>"Кроссовки", 2=>"Шорты");

$len\_mass=count($closets); *//вычисляет количество элементов*

for ($i=0; $i<$len\_mass; $i++)

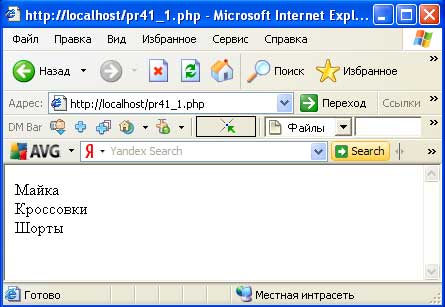
{

echo $closets[$i] . "<br>"; *// вывод элемента*

}

?>

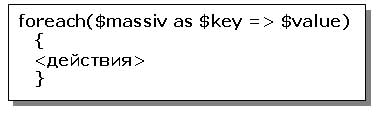
Результат выполнения программы вы можете увидеть на рисунке 2.

  
Рис.2. Результат работы скрипта

Ключ первого элемента в нашем массиве будет 0. Таким образом, мы проходим по всем элементам массива и выводим их. В этот цикл уже можно добавить определенное условие, предъявляемое к элементам массива.

**Обход массивов. Конструкция foreach**

Использованный в примере предыдущего шага массив является лишь частным случаем. На практике обычно встречаются массивы с непоследовательной индексацией. Поэтому используется специальная конструкция foreach, которая появилась только в четвертой версии PHP. В общем случае она выглядит так:

  
Рис.1. Общий вид конструкции foreach

В этой конструкции мы видим три переменные, среди которых $massiv - это просматриваемый массив. Переменные $key и $value содержат, соответственно, ключ и значение. Итак, поясним ход работы этого оператора цикла на примере:

<?php

$closets = array("Петров" => "Майка",

"Иванов" => "Кроссовки",

"Сидоров" => "Шорты");

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

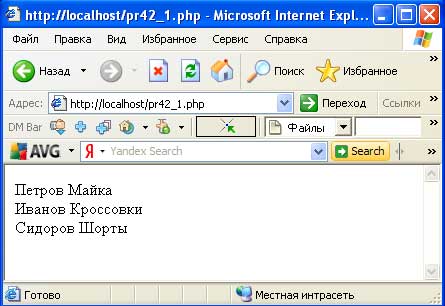
{

echo $key . " " . $value . "<br>"; *// вывод элемента*

}

?>

Результат выполнения программы вы можете увидеть на рисунке 2.

  
Рис.2. Результат работы скрипта

В первой строке мы создаем массив из трех элементов. Заметьте, что индексация производится с помощью строк. Далее следует конструкция, которая начинается со слова foreach. Затем в круглых скобках помещается рассматриваемый массив, после которого записывается специальное слово as. После него следует пара ключ-значение, разделяемая оператором =>. Итак, на первой итерации переменной $key присваивается ключ первого элемента, а переменной $value - его значение. При следующей итерации в переменные $key и $value запишутся соответственно ключ и значение следующего элемента. И так далее, пока массив не будет пройден полностью.

У вас может возникнуть вопрос о способе реализации этого цикла, так как механизм перемещения по массиву остается скрытым от программиста. Дело в том, что любой массив помимо ключей и значений имеет внутренний указатель (pointer) или курсор, с помощью которого можно узнать, какой элемент мы сейчас рассматриваем. Этим указателем, естественно, можно управлять, то есть переносить его от одного элемента к другому. Для работы с указателем применяются функции reset(), each(), list() и другие.

**Обход массивов. Функция reset()**

Как уже было сказано, для работы с указателем применяются функции reset(), [each()](http://it.kgsu.ru/PHP/php044.html), [list()](http://it.kgsu.ru/PHP/php045.html) и другие. Приведем краткий обзор этих функций.

*Функция* reset().

Ее общий вид:

http://it.kgsu.ru/PHP/images/ris43_1.jpg  
Рис.1. Общий вид функции reset()

Функция reset() устанавливает указатель на первый элемент массива, который является для нее входным параметром. Возвращает она значение первого элемента массива. Например:

<?php

$closets = array("Петров" => "Майка",

"Иванов" => "Кроссовки",

"Сидоров" => "Шорты");

echo reset($closets); *// выводит слово "Майка"*

?>

**Обход массивов. Функция each()**

*Функция* each().

Ее общий вид:

http://it.kgsu.ru/PHP/images/ris44_1.jpg  
Рис.1. Общий вид функции each()

Функция each() возвращает ключ и значение элемента, на который в данный момент указывает курсор. Причем ключ индексируется числом 0, а значение - 1. Затем функция смещает указатель на один элемент вправо. Например:

<pre>

<?php

$closets = array("Петров" => "Майка",

"Иванов" => "Кроссовки",

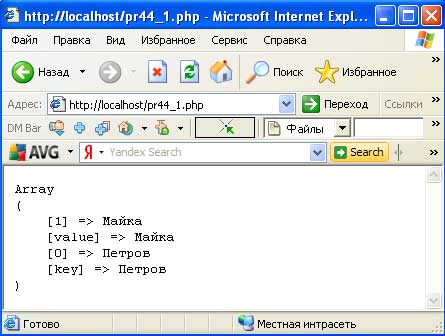
"Сидоров" => "Шорты");

print\_r(each($closets)); *// выводит массив*

?>

</pre>

Результат выполнения программы вы можете увидеть на рисунке 2.

  
Рис.2. Результат выполнения программы

**Обход массивов. Функция list()**

***Функция* list().**

Ее общий вид:

  
Рис.1. Общий вид функции **list()**

Поясним использование конструкции **list()** на примере.

**<?php**

**$closets = array(0 => "Майка",**

**1 => "Кроссовки",**

**2 => "Шорты");**

**list($thing\_1, $thing\_2, $thing\_3) = $closets;**

**echo $thing\_1; *// выводит "Майка"***

**echo "<br>";**

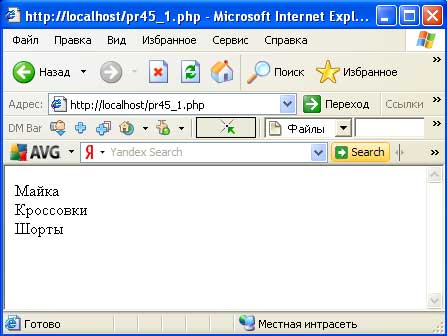
**echo $thing\_2; *// выводит "Кроссовки"***

**echo "<br>";**

**echo $thing\_3; *// выводит "Шорты"***

**?>**

Результат выполнения программы вы можете увидеть на рисунке 2.

  
Рис.2. Результат выполнения программы

Конструкция **list()** позволяет записать в переменные, которые находятся в круглых скобках, значения элементов массива. Но помните, что она работает с элементами, проиндексированными числами, причем, начиная строго с нуля.

**Обход массивов. Совместное использование функций reset(), each(), list()**

До появления цикла foreach функции reset(), each(), list() очень часто использовали вместе, чтобы просмотреть массив с непоследовательной индексацией. Например:

<?php

$closets = array("Петров" => "Майка",

"Иванов" => "Кроссовки",

"Сидоров" => "Шорты");

reset($closets); *// установка указателя на первый элемент*

while (list($key, $value) = each($closets))

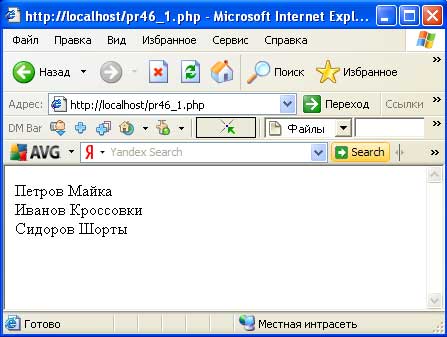
{

echo $key . " " . $value . "<br>"; *// вывод элемента*

}

?>

Результат выполнения программы вы можете увидеть на рисунке 1.

  
Рис.1. Результат выполнения программы

Не сложно догадаться, что этот пример по своей сути представляет собой цикл foreach. Сначала мы устанавливаем указатель на первый элемент массива с помощью функции count(). Затем в цикле начинаем последовательно выводить значения элементов вместе с их ключами. Обратите внимание на логическое выражение оператора while. Функция each() возвращает массив с данными для вывода, а конструкция list() записывает их в переменные $key и $value. Происходит это до тех пор, пока функция each() не возвратит пустой массив, что эквивалентно false.

Именно эту сложную конструкцию заменяет цикл foreach. Обратите внимание, что оператор foreach перед началом обхода тоже устанавливает указатель на первый элемент и не возвращает его обратно после завершения цикла.

Стоит отметить, что в конструкции foreach по желанию значение ключа можно опустить. Тогда мы будем работать только со значениями элементов.

<?php

$closets = array("Петров" => "Майка",

"Иванов" => "Кроссовки",

"Сидоров" => "Шорты");

foreach($closets as $value)

{

echo $value . "<br>"; *// вывод значения*

}

?>

В этом случае программа выведет слова *Майка, Кроссовки, Шорты*.

**Операторы работы с массивами. Сложение массивов**

Ранее мы рассматривали операторы применительно к переменным скалярного типа. Тем не менее, для массивов тоже существует рад операторов, которые можно использовать.

**Сложение массивов**

Сложение массивов происходит посредством оператора суммы "+". Например:

**<pre>**

**<?php**

**$closets\_1 = array("Петров"=>"Ботинки",**

**"Иванов" => "Футболка");**

**$closets\_2 = array("Петров"=>"Майка",**

**"Иванов" => "Кроссовки",**

**"Сидоров" => "Шорты");**

**$closets\_sum1 = $closets\_1 + $closets\_2;**

**print\_r($closets\_sum1);**

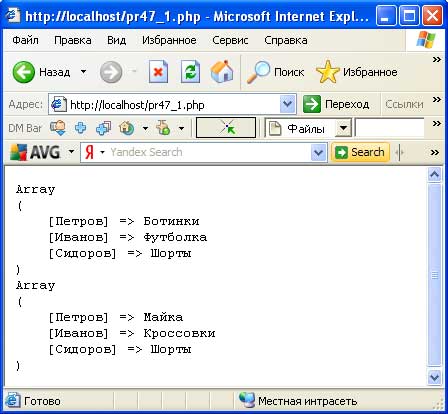
**$closets\_sum2 = $closets\_2 + $closets\_1;**

**print\_r($closets\_sum2);**

**?>**

**</pre>**

Результат выполнения программы вы можете увидеть на рисунке 1.

  
Рис.1. Результат выполнения программы

В первом случае мы видим, что результирующий массив содержит в себе два элемента первого и один элемент второго. Чтобы понять этот результат, нужно знать общий принцип сложения массивов. Результирующий массив состоит целиком из первого слагаемого, к которому добавляются элементы второго, имеющие отличные индексы. В связи с этим в первом случае отсутствуют элементы ***Майка*** и ***Кроссовки***, а во втором - ***Ботинки*** и ***Футболка***.

**Операторы работы с массивами. Сравнение массивов**

Сравнение массивов производится с помощью привычных для нас операторов отношений. Наибольший интерес в этом плане вызывают *операторы равенства* (==) и *эквивалентности* (===).

Массивы считаются равными, в том случае, если каждый элемент одного массива имеет один равный ему во втором, и наоборот. Равенство элементов подразумевает соответственно совпадение ключа и значения. Порядок расположения элементов при этом не играет роли. Например:

<?php

$closets\_1 = array ("Ботинки", "Футболка");

$closets\_2 = array("1" => "Футболка","0" => "Ботинки");

if ($closets\_1 == $closets\_2)

{

echo "Массивы равны";

}

else

{

echo "Массивы неравны";

}

?>

В этом случае программа выведет *"Массивы равны"*. Заметьте, что индексы второго массива находятся в двойных кавычках, а значит, имеют тип String. Но в данном случае сравниваются значения, а не типы, поэтому программа все равно выведет сообщение о равенстве массивов. То же самое относится к значениям элементов в массиве.

Оператор эквивалентности требует от своих операндов не только равенства значений, но и одинаковый порядок следования элементов в массиве. Например:

<?php

$closets\_1 = array("Ботинки","Футболка");

$closets\_2 = array("1" => "Футболка","0" => "Ботинки");

if ($closets\_1 === $closets\_2)

{

echo "Массивы эквивалентны";

}

else

{

echo "Массивы неэквивалентны";

}

?>

В этом случае программа выведет, что массивы неэквивалентны.

Наряду с разобранными операторами существуют также операторы неравенства (!=) и неэквивалентности (!==), принцип работы которых аналогичен.

**Модификация массивов. Добавление элементов массива**

На этом шаге мы рассмотрим ***функцию, добавляющую элементы в массив***.

Очень часто на практике требуется работать не просто со значениями, которые хранятся в массиве, а с его элементами в целом, то есть добавлять и удалять их.

Добавление элементов к массиву осуществляется функцией **array\_push()**, которая добавляет один или несколько элементов в конец массива. Общий вид этой функции следующий:

  
Рис.1. Общий вид функции **array\_push()**

Например:

**<pre>**

**<?php**

**$closets = array("Ботинки", "Футболка");**

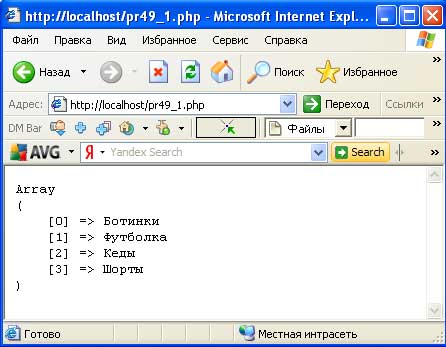
**array\_push($closets, "Кеды", "Шорты");**

**print\_r($closets);**

**?>**

**</pre>**

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Результат работы скрипта

**Модификация массивов. Удаление элементов массива. Функция unset()**

Удаление элементов массива можно производить различными способами. Во-первых, удалить элемент массива можно с помощью функции unset().

Общий вид этой функции следующий:

  
Рис.1. Общий вид функции unset()

На самом деле эта функция предназначена для уничтожения переменных, но на практике ее часто используют при работе с массивами. Например:

<pre>

<?php

$closets = array("Ботинки", "Футболка", "Шорты");

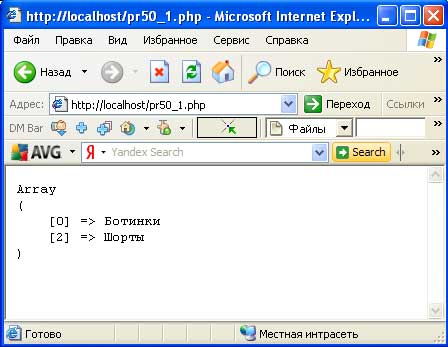
unset($closets[1]);

print\_r($closets);

?>

</pre>

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Результат работы скрипта

В этом примере мы удалили элемент с индексом 1.

**Модификация массивов. Удаление элементов массива.**

**Функция array\_pop()**

Подобно функции array\_push() есть функция array\_pop(), которая удаляет последний элемент массива.

Общий вид этой функции следующий:

http://it.kgsu.ru/PHP/images/ris51_1.jpg  
Рис.1. Общий вид функции array\_pop()

Например:

<pre>

<?php

$closets = array("Ботинки", "Футболка", "Шорты");

array\_pop($closets);

print\_r($closets);

?>

</pre>

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Результат работы скрипта

**Сортировка массивов. Функция sort()**

Самая простая и, наверное, самая распространенная функция сортировки - sort(). Она располагает элементы массива в алфавитном порядке.

Общий вид этой функции следующий:

http://it.kgsu.ru/PHP/images/ris52_1.jpg  
Рис.1. Общий вид функции sort()

Эта функция назначает новые ключи для элементов массива. Все ранее назначенные значения ключей будут удалены, вернее, переназначены.

Дополнительный второй параметр *тип сортировки* является необязательным, его можно использовать для изменения поведения сортировки. Этот параметр имеет следующие значения:

SORT\_REGULAR - сравнивать элементы нормально (не изменять типы);

SORT\_NUMERIC - сравнивать элементы в числовом отношении;

SORT\_STRING - сравнивать элементы как строки.

Например:

<pre>

<?php

$closets = array(0 => "Шорты", 1 => "Майка", 2 => "Кроссовки");

sort($closets);

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

{

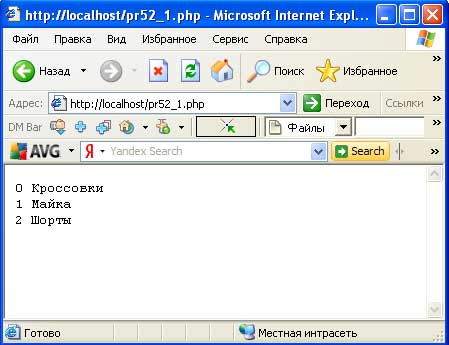
echo $key . " " . $value . "<br>"; *// вывод элемента*

}

?>

</pre>

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Результат работы скрипта

Теперь у нас элемент со значением *"Кроссовки"* имеет ключ 0, элемент со значением *"Майка"* имеет ключ 1, элемент со значением *"Шорты"* имеет ключ 2. На самом деле, какие бы у нас не были ключи до сортировки, функция sort() сама индексирует элементы, начиная с нуля. Например:

<pre>

<?php

$closets = array(3 => "Шорты", 4 => "Майка", 1 => "Кроссовки");

sort($closets);

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

{

echo $key . " " . $value . "<br>"; *// вывод элемента*

}

?>

</pre>

Сравните результат с приведенным на рисунке 2. Как видим, результат точно такой же, как и в первом случае. В связи с этим стоит быть осторожнее при сортировке массивов, где индексация имеет значение. Например, если вы в качестве ключей используете строки, то функция sort() все равно заменит их числами.

**Сортировка массивов. Второй параметр функции sort()**

Еще одна особенность функции **sort()** - необязательный второй параметр (флаг), который указывает на тип сортируемых значений. Например:

**<?php**

**$closets\_1 = array(0=>2, 1=>3, 2=>"Кроссовки");**

**sort($closets\_1, SORT\_NUMERIC);**

**foreach ($closets\_1 as $key => $value)**

**{**

**echo $key." ".$value."<br>"; *// вывод элементов***

**}**

**echo "<br>";**

**$closets\_2 = array(0=>2, 1=>3, 2=>"Кроссовки");**

**sort($closets\_2, SORT\_STRING);**

**foreach ($closets\_2 as $key => $value)**

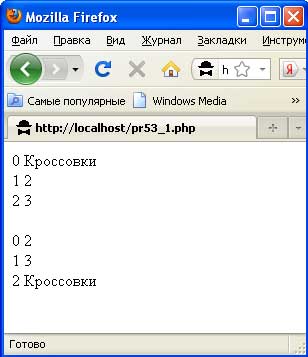
**{**

**echo $key." ".$value."<br>"; *// вывод элементов***

**}**

**?>**

Результат выполнения этой программы смотрите на рисунке 1.

  
Рис.1. Результат работы скрипта

В первом случае функция воспринимает все значения как числа, поэтому ***"Кроссовки"*** оказались на месте первого элемента (строка преобразовалась в число 0). Во втором случае значения сортируются как строки, поэтому на этот раз строка ***"Кроссовки"*** оказалась на месте последнего элемента. Существует так же третий флаг -**SORT\_REGULAR**, который указывает на обычную сортировку.

***Задание.****Попробуйте отсортировать массив с параметром****SORT\_REGULAR****и прокомментируйте полученный результат.*

Необязательные параметры функции **sort()** появились только в четвертой версии **PHP**.

**Сортировка массивов. Функция asort()**

Функцию **sort()** неудобно применять к массивам, где нужно сохранять значение ключей. Для решения этой проблемы применяют функцию **asort()**, которая работает по аналогичному принципу, но не изменяет индексы элементов. Например:

**<?php**

**$closets = array(3 => "Шорты", 4 => "Майка", 1 => "Кроссовки");**

**asort($closets);**

**foreach ($closets as $key => $value)**

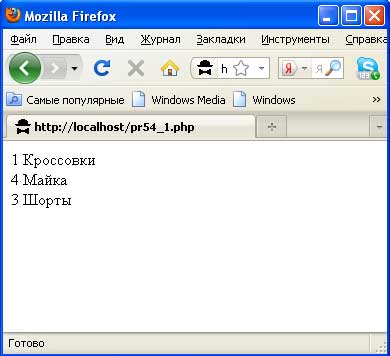
**{**

**echo $key . " " . $value . "<br>"; *// вывод элементов***

**}**

**?>**

Результат выполнения этой программы смотрите на рисунке 1.

  
Рис.1. Результат работы скрипта

Как видим, результаты сортировок одинаковые за исключением того, что индексы элементов остались прежними. Необязательные параметры также могут присутствовать в этой функции.

**Сортировка массивов. Функции rsort(), arsort(), ksort() и krsort()**

Часто встречаются случаи, когда надо отсортировать элементы массива в обратном порядке. Для этого применяются функции **rsort()** и **arsort()**, которые работают аналогично разобранным функциям.

Если есть сортировка элементов массива по значению, то логично было бы добавить сортировку по ключу. Именно эту задачу выполняет функция **ksort()**. Например:

**<?php**

**$closets = array("Петров" => "Майка",**

**"Иванов" => "Кроссовки",**

**"Сидоров" => "Шорты");**

**ksort($closets);**

**foreach ($closets as $key => $value)**

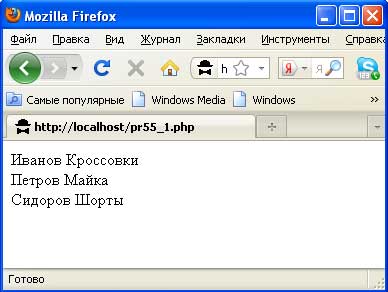
**{**

**echo $key . " " . $value . "<br>"; *// вывод элементов***

**}**

**?>**

Результат выполнения этой программы смотрите на рисунке 1.

  
Рис.1. Результат работы скрипта

Существует функция **krsort()**, которая, как вы поняли, сортирует массив в обратном порядке.

**Многомерные массивы**

До этого момента в качестве элементов массива мы использовали только числа и строки. На самом деле элементом массива может быть все что угодно, даже другие массивы. Выглядит это следующим образом:

$lang = array ("Петров" => array ("Английский", "Испанский", "Немецкий"),

"Иванов" => array ("Французский", "Итальянский"),

"Сидоров" => array ("Немецкий"));

Это пример инициализации двумерного массива, где в качестве элементов выступают обычные (одномерные) массивы. Он содержит информацию о знании иностранных языков. Мы можем еще более усложнить структуру, если в качестве элементов внутреннего массива возьмем еще один массив, тогда это уже будет трехмерный массив. При желании так можно продолжать очень долго, но на практике обычно останавливаются на трехмерном массиве.

Обращение к элементам внутреннего массива происходит следующим образом:

$lang["Петров"][0];

В данном случае мы обращаемся к первому элементу массива, который, в свою очередь, является первым элементом массива $lang, то есть к строке *Английский*.

Обход многомерных массивов можно также производить с помощью конструкции foreach. При этом мы будем иметь дело с вложенными циклами. Например:

<?php

$lang = array ("Петров" => array ("Английский", "Испанский", "Немецкий"),

"Иванов" => array ("Французский", "Итальянский"),

"Сидоров" => array ("Немецкий"));

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

{

echo $key."<br>"; *// вывод элементов*

foreach ($value as $sub\_key=>$sub\_value)

{

echo $sub\_key." ".$sub\_value."<br>"; *// вывод элементов*

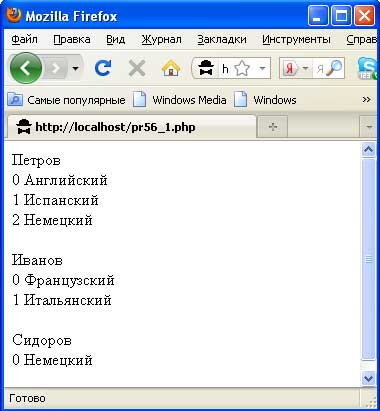
}

echo "<br>";

}

?>

Результат выполнения этой программы смотрите на рисунке 1.

  
Рис.1. Результат работы скрипта

Итак, во внешнем цикле переменная $key будет принимать значения индексов внешнего массива, то есть строки*Петров, Иванов, Сидоров*, а переменная $value - внутренние массивы. Переменные $sub\_key и $sub\_valueработают уже непосредственно с ключами и значениями внутренних массивов.

**Преобразование значений переменных в массив**

В PHP возможно преобразование типов данных в массив. Приведем пример преобразования известных нам типов данных в массив.

<pre>

<?php

$var\_int = 123;

$var\_double = 12.12;

$var\_string = "World";

$var\_boolean = TRUE;

print\_r((array)$var\_int);

echo "<br>";

print\_r((array)$var\_double);

echo "<br>";

print\_r((array)$var\_string);

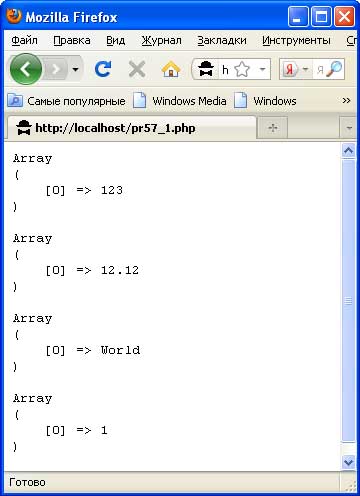
echo "<br>";

print\_r((array)$var\_boolean);

?>

</pre>

Результат выполнения этой программы смотрите на рисунке 1.

  
Рис.1. Результат работы скрипта

Во всех случаях создается массив с единственным элементом, ключ которого равен нулю, а значение соответствует преобразуемой переменной. Относительно других типов данных можно сказать, что преобразование происходит точно так же.

**Пример использования массивов**

Рассмотрим решение следующей задачи: найти и вывести номера всех отрицательных элементов одномерного массива.

Вот текст программы:

<html>

<head>

<title>Решение задачи</title>

</head>

<body>

Найти и вывести номера всех отрицательных элементов

одномерного массива.

<form>

Массив:<br><textarea name="str\_mas" cols="20" rows="5" wrap="virtual">

</textarea><br>

<input type="submit" name="go" value="Передать">

<input type="reset" value="Очистить">

</form>

<?php

$mas=explode(" ",$str\_mas);

echo "Массив:<br>";

print\_r($mas);

echo "<br>Номера отрицательных членов:<br>";

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

{

if ($value<0)

{

echo $key."<br>";

}

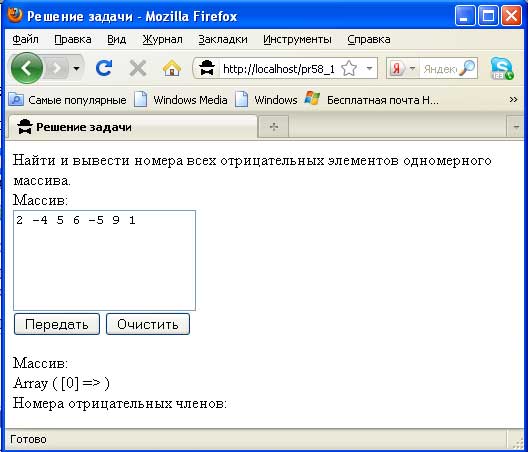
}

?>

</body>

</html>

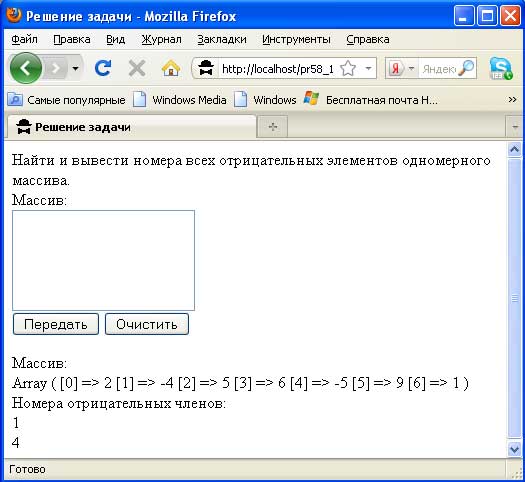
В окне браузера форма будет выглядеть так, как показано на рисунке 1 (естественно, без введенных значений):

  
Рис.1. Форма в окне браузера

Когда пользователь нажмет кнопку *"Передать"*, браузер передаст сценарию параметр str\_mas - значение текстовой области. Этот параметр передается так же как и значения из текстового поля, то есть для него создается переменная с соответствующим именем. В текстовую область будем вводить значения элементов массива через пробел.

Затем с помощью функции explode() преобразуем строку $str\_mas в массив $mas. После с помощью конструкцииforeach отбираем нужные нам элементы.

Ведем конкретные значения: str\_mas="2 -4 5 6 -5 9 1" (как показано на рисунке 1) и нажмем кнопку *"Передать"*. Результат работы программы представлен на рисунке 2.

  
Рис.2. Результат работы программы

**Строки. Определение строк**

В **РНР** существует скалярный тип данных **String**, переменные которого называют ***строками***. По своей сути строка - это набор символов.

Определить строку можно следующими способами:

**$stroka1 = "Hello, World!"; *// с помощью двойных кавычек***

**$stroka2 = 'Hello, World!'; *// с помощью одинарных кавычек***

В данном случае строка создается с помощью оператора присваивания и двойных или одинарных кавычек. То есть определение строки почти ни чем не отличается от определения переменных других скалярных типов. Тем не менее, проблемы начинают возникать, например, если вам требуется вывести одинарную или двойную кавычку. Например:

**<?php**

**$str = 'Он сказал: 'Привет!' '; *// эта строка вызовет ошибку***

**echo $str;**

**?>**

В данном случае в окне браузера будет выведено сообщение об ошибке, которая заключается в том, что неправильно определена строка. Это происходит потому, что строкой считается набор символов между первой и ближайшей к ней кавычкой. Для решения этой проблемы применяют экранирование символов с помощью обратной косой черты (\). Тогда наша программа будет выглядеть так:

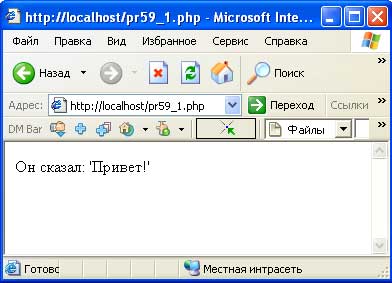
**<?php**

**$str = 'Он сказал: \'Привет!\' ';**

**echo $str;**

**?>**

Результат выполнения этой программы смотрите на рисунке 1.

  
Рис.1. Пример вывода строки

**Строки. Определение строк (окончание)**

На практике часто требуется, чтобы строки содержали в себе также управляющие символы, такие, как перевод строки, табуляцию, другие символы форматирования текста. Но такие последовательности можно помещать только в двойные кавычки:

**<?php**

**echo 'Одинарные кавычки \n';**

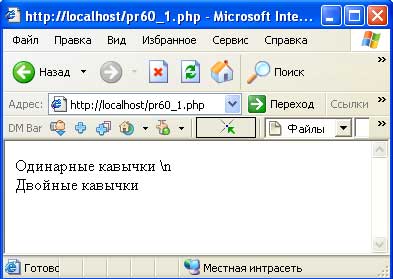
***// выводит: Одинарные кавычки \n***

**echo "<br>";**

**echo "Двойные кавычки \n"; *// выводит: Двойные кавычки***

**?>**

Результат выполнения этой программы смотрите на рисунке 1.

  
Рис.1. Пример вывода строки

Несложно заметить, что в строках с одинарными кавычками специальные символы не распознаются, и это не единственное различие.

Одна из самых главных особенностей строк, определенных с помощью двойных кавычек, - это возможность обрабатывать переменные внутри них.

**<?php**

**$str = "Иван";**

**echo "Привет, $str!"; *// выведет: Привет, Иван!***

**?>**

В данном случае вместо подстроки **$str** подставляется значение переменной **$str**. Происходит это по следующей схеме. Подстрока будет считаться переменной, если она образует правильное имя переменной. Например, в строке**Hello, $strs!** переменной будет считаться **$strs**, а не **$str** как в предыдущем примере. Если все-таки требуется распознать именно переменную **$str** в этой строке, то надо ее заключить в фигурные скобки:

**<?php**

**$str = 'Ivan';**

**echo "Hello, {$str}s!"; *// выведет: Hello, Ivans!***

**?>**

В том случае если вам понадобится вывести знак доллара (например, для вывода имени переменной), можно использовать одинарные кавычки или экранировать его:

**<?php**

**$str = "Hello";**

**echo 'Переменная имеет имя $str';**

***// выведет: Переменная имеет имя $str***

**echo "<br>";**

**echo "Переменная имеет имя \$str";**

***// выведет: Переменная имеет имя $str***

**?>**

В обоих случаях переменная в строках не определяется.

**Строки. Вывод строк. Конструкция echo**

В РНР имеется много способов вывода строк. Для начала разберем самый простой из них - конструкцию echo.

Конструкцию echo можно использовать как с круглыми скобками, так и без них. Просто во всех примерах подчеркивается тот факт, что echo является конструкцией языка, а не функцией.

Еще одна особенность echo заключается в том, что с помощью нее можно выводить несколько сообщений:

<?php

$name = "Анна";

echo "Привет, ".$name."!"; *// выводит "Привет, Анна!"*

?>

**Строки. Вывод строк. Функция printf()**

До этого мы выводили строки в окне браузера с помощью команды echo. Однако в РНР имеется еще несколько способов, например с помощью функции printf().

На рисунке 1 приведен общий вид этой функции:

  
Рис.1. Общий вид функции printf()

Она выводит строку в определенном формате, который задает программист. Например:

<?php

$str = "Число 8 в двоичном представлении: %b";

printf($str, 8);

***// выводит: Число 8 в двоичном представлении: 1000***

?>

Первым аргументом функции printf() является строка для вывода. Ее формат определяется с помощью сочетания специальных символов. В данном случае это символ "%", который всегда ставится первым, и буква b, определяющая двоичный формат вывода целого числа. Это число передается в качестве следующего входного параметра. Заметьте, что сначала оно преобразуется к целому числу, а затем выводится в двоичном представлении.

В строке имеется возможность вставить несколько различных комбинаций специальных символов, при этом их количество должно совпадать с числом параметров функции printf().

<?php

$str = "Двоичное и восьмеричное представление числа 12: %b и %o";

/\* выведет: Двоичное и восьмеричное представление числа 12: 1100 и 14 \*/

printf($str, 12, 12);

?>

Буква "o" указывает на вывод целого числа в восьмеричном представлении. Полный список специальных символов представлен в таблице 1.

|  |  |
| --- | --- |
| Таблица 1. Специальные символы | |
| **Символ** | **Описание** |
| **b** | Параметр преобразуется в целое и выводится в виде двоичного числа |
| **с** | Параметр преобразуется в целое и выводится в виде символа с соответствующим кодом ASCII |
| **d** | Параметр преобразуется в целое и выводится в виде десятичного числа со знаком |
| **e** | Параметр преобразуется в вещественное число и выводится в научной нотации (например: 1.2e+2) |
| **u** | Параметр преобразуется в целое и выводится в виде десятичного числа без знака |
| **f** | Параметр преобразуется в вещественное число и выводится в виде двоичного числа |
| **о** | Параметр преобразуется в целое и выводится в виде восьмеричного числа |
| **s** | Параметр преобразуется в строку |
| **x** | Параметр преобразуется в целое и выводится в виде шестнадцатеричного числа (в нижнем регистре) |
| **X** | Параметр преобразуется в целое и выводится в виде шестнадцатеричного числа (в верхнем регистре) |

Также имеется возможность выравнивания текста. Например:

<pre>

<?php

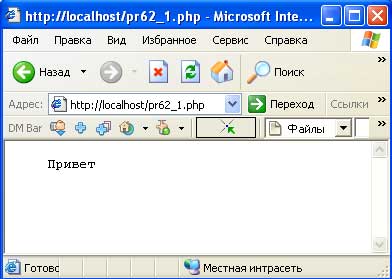
$str = "%10s";

printf($str, "Привет");

?>

</pre>

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Пример вывода строки

Обратите внимание, что выравнивание текста происходит по правому краю поля. Таким способом очень удобно выводить табличные данные.

Если требуется выравнивание по левому краю, то нужно использовать такую же запись, но со знаком минус (-):

<?php

$str = "%-10s";

printf($str, "Привет");

?>

</pre>

Помимо этого, с помощью функции printf() можно задавать точность выводимого числа, автоматически дополнять недостающие символы и многое другое.

**Строки. Длина строки**

На практике очень часто требуется узнать длину строки. Для этого в РНР имеется специальная функция strlen(), принимающая в качестве входных параметров строку и возвращающая ее длину виде целого числа.

Ее общий вид следующий:

http://it.kgsu.ru/PHP/images/ris63_1.jpg  
Рис.1. Общий вид функции strlen()

Например:

<?php

$str = "Hello, World!";

$len = strlen($str); ***// длина строки***

for ($i=0; $i<=$len; $i++) ***// посимвольный вывод строки***

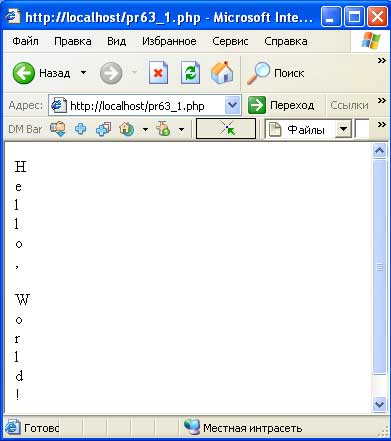
{echo $str[$i];

echo "<br>";

}

?>

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Пример посимвольного вывода строки

Первое, что бросается в глаза - это присутствие квадратных скобок, так, как они использовались при работе с массивами. Здесь нет ничего странного, так как в РНР, как и во многих других языках программирования, строку можно воспринимать как массив символов, проиндексированный целыми числами, начиная с нуля.

Вернемся к нашему примеру. Сначала мы инициализируем строку, затем с помощью функции strlen() получаем ее длину в виде целочисленного значения, которое запишем в переменную $len. Далее создаем простой цикл for на $lenитераций. У многих начинающих программистов часто возникает соблазн не использовать дополнительную переменную для хранения длины строки. Действительно, функцию strlen() можно записать прямо в определении цикла for. Например:

for ($i=0;$i<= strlen($str);$i++) {...}.

Тем не менее, категорически не рекомендуется так делать. Во-первых, этот код работает медленнее, чем с использованием дополнительной переменной, так как приходится постоянно вызывать функцию strlen(). Во-вторых, длина строки может меняться в теле цикла, соответственно, поменяется количество итераций, что может повлиять на ход программы.

В теле цикла нашей программы мы выводим один символ и выполняем перевод строки. Еще раз хочется обратить ваше внимание, что нумерация символов происходит с нуля.

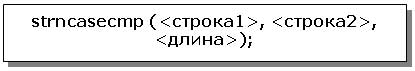
**Строки. Сравнение строк**

Функции **strcasecmp()** и **strncasecmp()** позволяют производить сравнение строки определенного количества символов. Чтобы произвести сравнение двух строк, используют функцию **strcasecmp()**. Она позволяет сравнивать строки без учета регистра. Ее синтаксис:

http://it.kgsu.ru/PHP/images/ris64_1.jpg  
Рис.1. Общий вид функции **strcasecmp()**

Параметры ***строка1*** и ***строка2*** задают значения сравниваемых строк. Функция возвращает целое число, равное нулю, если две строки равны. Если ***строка1*** меньше чем ***строка2***, то функция возвратит целое значение меньше нуля. Если ***строка1*** будет больше ***строка2***, функция возвратит целое значение больше нуля.

Если нужно сравнивать не всю строку сразу, а определенное начальное количество символов этой строки, применяют функцию **strncasecmp()**. Она имеет следующий синтаксис:

  
Рис.2. Общий вид функции **strncasecmp()**

На месте параметра ***длина*** указывается количество символов, отсчитываемых с начала строк ***строка1*** и ***строка2***. Это целое число. Во всем остальном функция **strncasecmp()** работает аналогично функции ***strcasecmp()***. Приведем пример, иллюстрирующий использование функций **strcasecmp()** и **strncasecmp()**:

**<?php**

**$var1 = "TABLE";**

**$var2 = "table";**

**if (!strcasecmp ($var1, $var2))**

**{**

**echo "Строки $var1 и $var2 равны без учета регистра"."<br>";**

**}**

**else**

**{**

**echo "Строки $var1 и $var2 не равны без учета регистра"."<br>";**

**}**

**$var3 = "World";**

**$var4 = "Hello";**

**if ($int=strcasecmp($var3,$var4))**

**{**

**echo "Строки $var3 и $var4 равны"."<br>";**

**}**

**else**

**{**

**echo "Строки $var3 и $var4 не равны"."<br>";**

**}**

**$var5 = "World";**

**$var6 = "He";**

**if ($int=strcasecmp($var5,$var6))**

**{**

**echo "Строки $var5 и $var6 равны"."<br>";**

**}**

**else**

**{**

**echo "Строки $var5 и $var6 не равны"."<br>";**

**}**

**$var7 = "Vitalik";**

**$var8 = "Vitya";**

**if (!strncasecmp($var7,$var8,3))**

**{**

**echo "Первые три символа строк $var7 и $var8 равны"."<br>";**

**}**

**else**

**{**

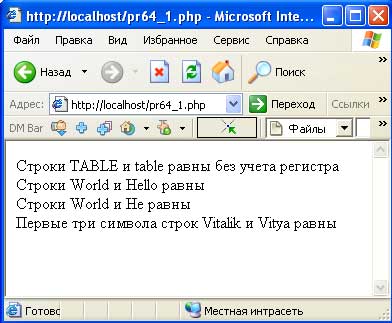
**echo "Первые три символа строк $var7 и $var8 не равны"."<br>";**

**}**

**?>**

В первом фрагменте приведенного примера происходит сравнение двух строк, одна из которой приведена в верхнем регистре, другая - в нижнем. Во второй части программы происходит сравнение двух строк с одинаковой длиной, но разными значениями. Функция сравнивает каждый символ этих строк, и за счет того, что сравниваемые символы будут различны, функция выдаст соответствующий результат. В третьей части программы происходит обычное сравнение количества символов строк, только эти строки имеют разные длины. Четвертая часть - пример сравнения строк при помощи функции **strncasecmp()**.

Результат выполнения этой программы смотрите на рисунке 3.

  
Рис.3. Пример применения функций **strcasecmp()** и **strncasecmp()**

Каждый отдельный блок скрипта выводит одну строку согласно полученному результату.

**Строки. Копирование части строки**

Очень часто при работе со строками программисту необходимо получить какую-либо определенную последовательность символов имеющейся строки. Для этого используется функция **substr()**. Она позволяет получать любую часть строки с учетом указанных параметров.

Ее общий вид:

http://it.kgsu.ru/PHP/images/ris65_1.jpg  
Рис.1. Общий вид функции **substr()**

Функция возвращает часть символов строки ***строка***, определяемую параметрами ***номер*** (с какого символа) и***длина*** (сколько символов будет в возвращаемой строке). Параметр ***длина*** является необязательным.

Если параметр ***номер*** положительный, то возвращаемая строка будет начинаться с символа строки с номером***номер***.

**<?php**

**$rest = substr("abcdef", 1); *// возвращает "bcdef"***

**echo $rest."<br>";**

**$rest = substr("abcdef", 1, 3); *// возвращает "bcd"***

**echo $rest."<br>";**

**$rest = substr("abcdef", 0, 4); *// возвращает "abcd"***

**echo $rest."<br>";**

**$rest = substr("abcdef", 0, 8); *// возвращает "abcdef"***

**echo $rest."<br>";**

***/\* к отдельным символам можно обращаться с помощью фигурных скобок \*/***

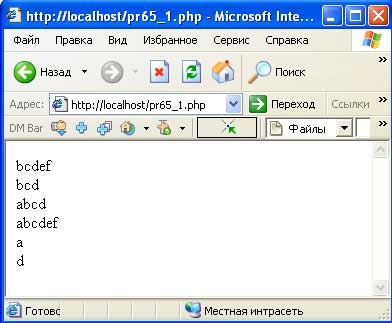
**$string = 'abcdef';**

**echo $string[0]."<br>"; *// выводит a***

**echo $string[3]."<br>"; *// выводит d***

**?>**

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Пример применения функции **substr()**

Если параметр ***номер*** отрицательный, то возвращаемая строка будет начинаться с символа ***номер*** от конца строки.

**<?php**

**$rest = substr("abcdef", -1); *// возвращает "f"***

**echo $rest."<br>";**

**$rest = substr("abcdef", -2); *// возвращает "ef"***

**echo $rest."<br>";**

**$rest = substr("abcdef", -3, 1); *// возвращает "d"***

**echo $rest."<br>";**

**?>**

Если параметр ***длина*** указан и он положительный, то возвращаемая строка закончится через ***длина*** символов от символа с номером ***номер***. Если параметр ***длина*** указан и он отрицательный, то возвращаемая строка закончится через ***длина*** от конца строки.

**<?php**

**$rest = substr("abcdef",0,-1); *// возвращает "abcde"***

**echo $rest."<br>";**

**$rest = substr("abcdef",2,-1); *// возвращает "cde"***

**echo $rest."<br>";**

**$rest = substr("abcdef",4,-4); *// возвращает "" (пустая строка)***

**echo $rest."<br>";**

**$rest = substr("abcdef",-3,-1); *// возвращает "de"***

**echo $rest."<br>";**

**?>**

**Строки. Поиск подстроки в строке**

Иногда бывает нужным выяснить, содержится ли подстрока в строке. Например, есть список фамилий, среди которых требуется найти Иванова. Осуществить это можно с помощью функции strstr().

Общий вид этой функции следующий:

http://it.kgsu.ru/PHP/images/ris66_1.jpg  
Рис.1. Общий вид функции strstr()

Принцип ее работы рассмотрим на примере:

<?php

$str = "Петров, Иванов, Сидоров";

$sub = "Иванов";

if (!strstr($str, $sub))

{

echo "Фамилия не найдена";

}

else

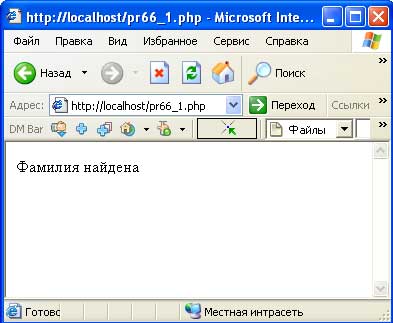
{

echo "Фамилия найдена";

}

?>

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Пример применения функции strstr()

Первый параметр функции является строкой, в которой производится поиск подстроки, которая, в свою очередь, передается вторым параметром. В нашем примере строкой является переменная $str, а подстрокой - $sub. Если результат поиска отрицательный, то функция возвращает значение false. Если совпадение имеется, то функция возвращает часть строки, начинающуюся с найденной подстроки.

Функция strstr() учитывает регистр символов. Для поиска без учета регистра используйте stristr(). Синтаксис этой функции такой же, как и функции strstr(). Например:

<?php

$email = 'USER@EXAMPLE.com';

$domain = stristr($email, 'e');

echo $domain; ***// выводит ER@EXAMPLE.com***

?>

**Строки. Функции удаления лишних пробелов. Функция chop()**

Когда пользователь вводит данные, очень часто он ставит лишние пробелы.

Чтобы произвести удаление лишних пробелов в обрабатываемой строке, необходимо воспользоваться функцией**chop()**.

Ее общий вид:

http://it.kgsu.ru/PHP/images/ris67_1.jpg  
Рис.1. Общий вид функции **chop()**

Прежде всего, разберемся, что значит лишние пробелы и в каком случае они действительно являются лишними.

Строка представляет собой совокупность символов, разделенных пробелами. Правильной считается строка, когда между каждым словом установлено не более одного пробела, за исключением условий, когда это действительно необходимо (в этом случае о применении функции **chop()** речи не может идти вообще). Эта функция удаляет все повторяющиеся пробелы между словами.

Функция **chop()** возвращает "правильную строку", т.е. строку, в которой будут удалены все лишние пробелы.

Рассмотрим пример работы функции **chop()**:

**<?php**

**$name = " Удалено два пробела, далее - три !!!";**

**$name1 = "Самая обычная строка";**

**$line = chop($name);**

**$line1 = chop($name1);**

**$line2 = chop("ввв ввв ввв ввв ввв ввв ");**

**echo $line."<br>";**

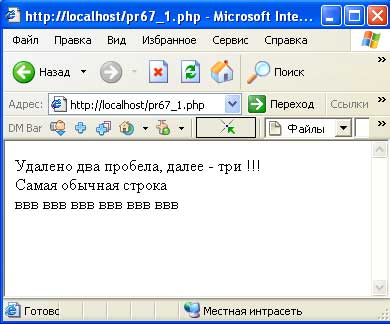
**echo $line1."<br>";**

**echo $line2."<br>";**

**?>**

В данном примере приведено три случая. Первый - это строка с большим количеством пробелов. Эту строку содержит переменная **$name**. Далее - обычная строка. То есть никакой ошибки не произойдет, если в качестве параметра будет задана правильная строка. В этом случае она просто произведет поиск лишних пробелов и вернет ее в строку **$line1**. В третьем случае строка указывается в самой функции.

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Пример применения функции **chop()**

**Строки. Функции удаления лишних пробелов. Функции trim(), ltrim() и rtrim()**

Иногда необходимо произвести удаление лишних пробелов с конца или с начала строки или же одновременно и с конца и с начала.

Для удаления символов пробела с начала и конца строки имеется функция trim().

Ее общий вид следующий:

  
Рис.1. Общий вид функции trim()

Например:

<?php

$str = " Иванов Иван Иванович ";

$new\_str = trim($str);

echo $new\_str; *// выводит "Иванов Иван Иванович"*

?>

В данном примере функция trim() возвращает строку без удаленных пробелов.

Второй параметр этой функции является необязательным и если он не задан, то удаляются следующие символы:

" " (ASCII 32 (0x20)) - символ пробела.

"\t" (ASCII 9 (0x09)) - символ табуляции.

"\n" (ASCII 10 (0x0A)) - символ перевода строки.

"\r" (ASCII 13 (0x0D)) - символ возврата каретки.

"\0" (ASCII 0 (0x00)) - NUL-байт.

"\x0B" (ASCII 11 (0x0B)) - вертикальная табуляция.

Можно также задать список символов для удаления. С помощью ".." можно задать диапазон символов. Например:

<pre>

<?php

$text = "\t\tЛишь несколько слов :) ... ";

echo trim($text)."<br>"; *// "Лишь несколько слов :) ..."*

echo trim($text," \t."); *// "Лишь несколько слов :)"*

*/\* удалить управляющие символы ASCII из начала и*

*конца строки $binary (от 0 до 31 включительно) \*/*

$clean = trim($binary, "\x00..\x1F");

?>

</pre>

    Так же есть функции ltrim() и rtrim() синтаксис которых аналогичен синтаксису функции trim().

    Функция ltrim() возвращает строку с удаленными из начала строки пробелами. Функция rtrim() возвращает строку с удаленными с конца строки пробелами. В остальном эти функции аналогичны функции trim(). Например:

<pre>

<?php

$text = "\t\tЛишь несколько слов :) ... ";

$trimmed = ltrim($text);

*// $trimmed = "Лишь несколько слов :) ... "*

echo $trimmed."<br>";

$trimmed = ltrim($text, " \t.");

*// $trimmed = "Лишь несколько слов :) ... "*

echo $trimmed."<br>";

$clean = ltrim($binary, "\x00..\x1F");

*/\* удалить управляющие символы ASCII из начала строки*

*$binary (от 0 до 31 включительно) \*/*

$text = "\t\tЛишь несколько слов :) ... ";

$trimmed = rtrim($text);

*// $trimmed = "\t\tЛишь несколько слов :) ..."*

echo $trimmed."<br>";

$trimmed = rtrim($text, " \t.");

*// $trimmed = "\t\tЛишь несколько слов :)"*

echo $trimmed."<br>";

$clean = ltrim($binary, "\x00..\x1F");

*/\* удалить управляющие символы ASCII из конца строки*

*$binary (от 0 до 31 включительно) \*/*

?>

</pre>

**Строки. Разбиение и соединение строк. Функция explode()**

В РНР имеется возможность производить разнообразные преобразования над строками. К этим преобразованиям также можно отнести разбиение и соединение строк.

Функция explode() разбивает строку на подстроки. Ее общий вид следующий:

  
Рис.1. Общий вид функции explode()

Чтобы разбить заданную строку необходимо указать, относительно чего будет осуществляться разделение. Например, возможно множество вариантов: относительно пробелов, символов и т. д.

Функция explode() возвращает массив значений, полученных при делении данной строки. Сама строка указывается вторым параметром, *символ* является как раз той точкой, относительно которой и будет происходить процесс деления. В этом вы убедитесь, изучив приведенный пример. Если установлен параметр *длина*, то массив будет содержать максимум *длина* элементов, при этом последний элемент будет содержать остаток строки.

Приведем пример:

<pre>

<?php

***// первый блок***

$part = "Часть1 Часть2 Часть3 Часть4 Часть5 Часть6 Часть7";

$piec = explode(" ",$part);

print\_r($piec);

echo "<br>";

***// второй блок***

$part1 = "Часть1 Часть2 Часть3 Часть4 Часть5 Часть6 Часть7";

$piec1 = explode("т",$part1);

print\_r($piec1);

echo "<br>";

***// третий блок***

$part2 = "Часть1 Часть2 Часть3 Часть4 Часть5 Часть6 часть7";

$piec2 = explode("ц",$part2);

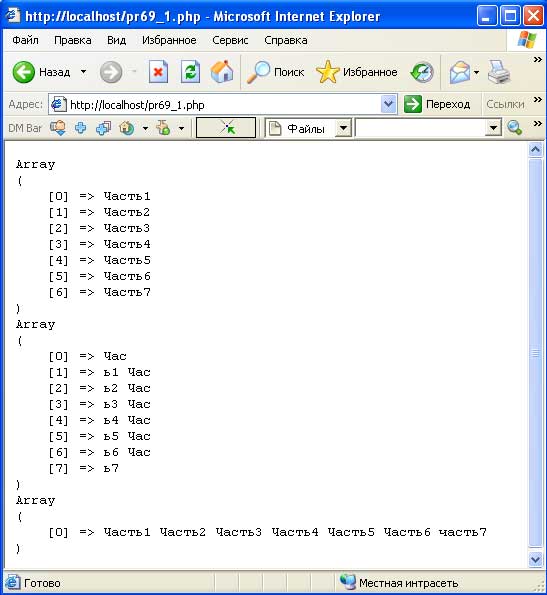
print\_r($piec2);

echo "<br>";

?>

</pre>

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Пример применения функции explode()

Данную программу можно разделить на три блока. В первом блоке происходит разделение строки на подстроки по пробелу. Функция explode() будет осуществлять исследование строки, проверяя последовательно каждый символ. Как только она найдет пробел, автоматически произойдет перенос указанного участка символов в массив. Затем функция продолжает проверять строку. После того как найдены все пробелы, будет создан массив символов разделенного предложения.

Во втором блоке происходит практически то же самое, только функция осуществляет разделение не по пробелу, а по букве "т". Это приведено для того, чтобы показать, что предложение можно разделить не только по словам, но и по любому другому символу. Обратите внимание, что самого символа (в данном случае "т") нет. В процессе деления функция explode() просто удаляет тот символ, относительно которого осуществляется деление строки. Другими словами, после того как функция удаляет символ, осуществляется процесс записи содержимого массива.

Третий блок программы приведен для того, чтобы вы могли сравнить, что произойдет, если функция не найдет указанного символа.

**Строки. Разбиение и соединение строк. Функция implode()**

Теперь осуществим обратный процесс. Все, что мы делали при помощи функции **explode()**, вернем обратно, т.е. соединим элементы полученного ранее массива в строку. Для решения этой задачи используем функцию **implode()**. Она имеет следующий синтаксис:

http://it.kgsu.ru/PHP/images/ris70_1.jpg  
Рис.1. Общий вид функции **implode()**

В качестве параметра ***символ*** необходимо подставлять символ, который будет соединять элементы массива. Все аналогично функции **explode()**, только в обратном порядке. Параметр ***имя массива*** отвечает за массив, элементы которого мы собираемся соединить.

Приведем пример:

**<pre>**

**<?php**

**echo "Осуществляется процесс деления строки при помощи функиии explode()"."<br>";**

**$part = "Часть1 Часть2 Часть3 Часть4 Часть5 Часть6 Часть7";**

**$piec = explode(" ",$part);**

**print\_r($piec);**

**echo "<br>";**

**$part1 = "Часть1 Часть2 Часть3 Часть4 Часть5 Часть6 Часть7";**

**$piec1 = explode("т",$part1);**

**print\_r($piec1);**

**echo "<br>";**

**$part2 = "Часть1 Часть2 Часть3 Часть4 Часть5 Часть6 часть7";**

**$piec2 = explode("ц",$part2);**

**print\_r($piec2);**

**echo "<br>"."<br>";**

**echo "Осуществляется процесс соединения при помощи функции implode()"."<br>";**

**$string = implode(" ",$piec);**

**$string1 = implode("т",$piec1);**

**$string2 = implode("ц",$piec2);**

**echo $string."<br>";**

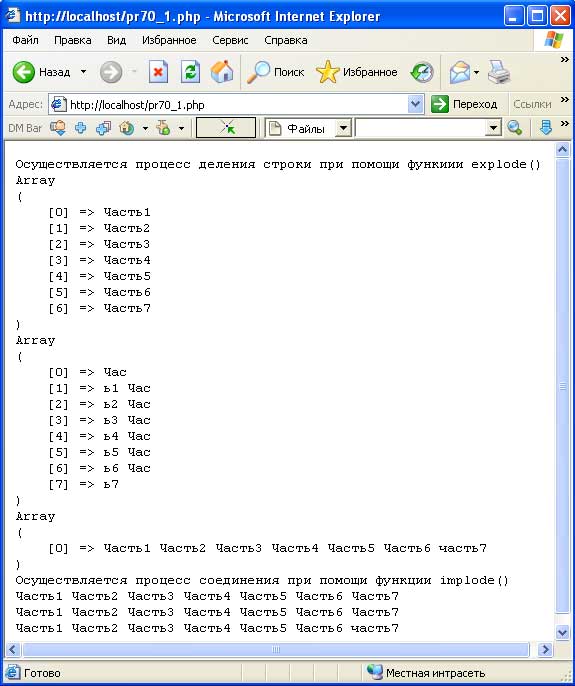
**echo $string1."<br>";**

**echo $string2."<br>";**

**?>**

**</pre>**

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Пример применения функции **implode()**

В начале примера осуществляется процесс разделения строки на массив символов при помощи функции **explode()**, затем при помощи функции **implode()** выполняется обратный процесс. Как видно, все три строки приняли первоначальный вид.

**Строки. Разбиение и соединение строк. Функция strtok()**

Рассмотрим следующую функцию:

http://it.kgsu.ru/PHP/images/ris71_1.jpg  
Рис.1. Общий вид функции **strtok()**

Эта функция разбивает строку ***строка*** на две подстроки, используя в качестве разделителей символы из параметра ***символы***.

Первый параметр функции **strtok()** указывает строку, которую нужно разделить при помощи функции **strtok()**, а второй параметр указывает непосредственно разделитель (или разделители).

Следует отметить, что только первый вызов функции **strtok()** использует параметр ***строка***. Для каждого последующего вызова функции **strtok()** необходим только разделитель. Чтобы разбить новую строку нужно снова вызвать **strtok()** с двумя параметрами.

Можно указывать несколько разделителей в параметр. Строка разделится при обнаружении любого из указанных символов.

Рассмотрим пример:

**<?php**

**$part = "Часть1 Часть2 Часть3 Часть4 Часть5 Часть6 Часть7";**

**echo $part."<br>";**

**$piec = strtok($part," ");**

**while ($piec)**

**{**

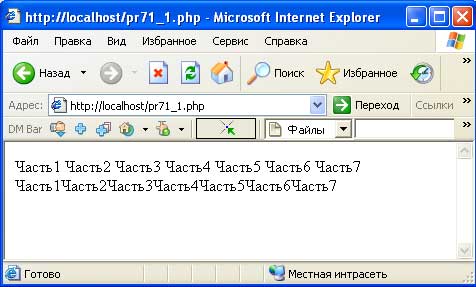
**echo $piec;**

**$piec = strtok(" ");**

**}**

**?>**

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Пример применения функции **strtok()**

Первая совокупность представляет собой строку. А вторая - последовательность отдельных слов, которые не являются единой строкой. Таким способом можно осуществлять любое разбиение строки с учетом необходимого разделителя.

Со следующего шага мы начнем знакомиться ***с операциями, выполняемыми над отдельными символами строки***.

**Строки. Операции с отдельными символами строки.**

**Функции strpos() и strrpos()**

Рассмотрим способы получения позиции символов при помощи функций strpos() и strrpos().

Функция strpos() возвращает позицию первого вхождения нужного символа. Приведем синтаксис функцииstrpos():

http://it.kgsu.ru/PHP/images/ris72_1.jpg  
Рис.1. Общий вид функции strpos()

Параметр *строка* устанавливает строку, в которой будет осуществляться поиск указанного символа. Сам символ устанавливается параметром *символ*.

Если в качестве параметра *символ* указывается строка, то использован будет только первый символ этой строки.

Если *символ* не найден, то возвращается false. Если второй параметр не является символом или строкой, то он рассматривается как десятичное число.

Третий параметр *количество* является необязательным. Он показывает, с какой позиции будет производиться поиск указанного символа. В этом случае длина, возвращенная функцией strpos(), все равно будет отсчитываться с начала строки. Приведем пример:

<?php

$string = "Мама купила красную шапку";

$strnew = strpos ($string,"п");

$strnew1 = strpos($string,"п",10);

echo "Первоначальная строка: ".$string."<br>";

echo "Количество символов, начиная с начала строки до первого символа п: ".

$strnew."<br>";

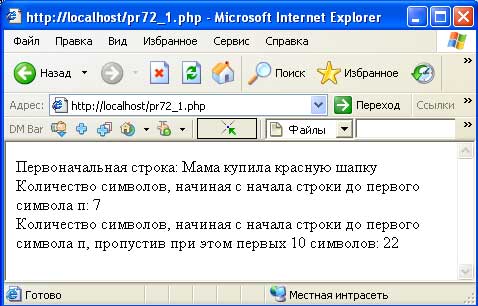
echo "Количество символов, начиная с начала строки до первого символа п,

пропустив при этом первых 10 символов: ".$strnew1."<br>";

?>

В первом случае результатом выполнения функции будет количество символов, отсчитанных с начала указанной строки, при этом третий параметр указан не был. Во втором случае был указан третий параметр, в результате чего поиск символа "п" будет начинаться не с начала строки, а с десятого символа. Поиск будет осуществляться до первого символа "п", после чего функция подсчитает количество символов до найденного символа.

Результат выполнения этой программы смотрите на рисунке 2.

  
Рис.2. Пример применения функции strpos()

Функция strrpos() по принципу работы аналогична функции strpos(), единственное ее отличие заключается в том, что функция strrpos() находит позицию последнего появления нужного нам символа в строке. Функция имеет следующий синтаксис:

http://it.kgsu.ru/PHP/images/ris72_3.jpg  
Рис.3. Общий вид функции strrpos()

Как было уже сказано, strrpos() возвращает номер позиции последнего появления символа *символ* в строке*строка*.

**Строки. Операции с отдельными символами строки.**

**Функции chr() и ord()**

Следующая функция, которую мы рассмотрим, - это функция **chr()**. Вот ее общий вид:

http://it.kgsu.ru/PHP/images/ris73_1.jpg  
Рис.1. Общий вид функции **chr()**

Она возвращает строку, состоящую из символа, код которого равен ***код символа*** в соответствии с таблицей**ASCII**.

Функция **ord()** возвращает код символа ***символ*** так же в соответствии с таблицей **ASCII**.

http://it.kgsu.ru/PHP/images/ris73_2.jpg  
Рис.2. Общий вид функции **ord()**

Приведем пример использования функций **chr()** и **ord()**:

**<?php**

**echo chr(75) ."<br>"; *// выведет букву "K"***

**echo ord("A"); *// выведет число 65***

**?>**

**Строки. Перевод строк в верхний и нижний регистры**

В каждой стране мира существуют свои стандарты письма. Но нет ни одной страны, где можно было бы обходиться без слов с большой буквы. Существуют даже ситуации, когда слово, находящее в середине предложения, должно начинаться с большой буквы (например, существительное в немецком языке).

Объясним, что значит верхний и нижний регистры. Когда вы используете строку только с заглавными буквами, это называется ***верхним регистром***. В противном случае - ***нижним регистром***. В **РНР** для изменения регистра используются следующие функции: **strtolower(), strtoupper(), ucfirst(), ucwords()**.

Все приведенные функции по своей структуре имеют аналогичный синтаксис:

http://it.kgsu.ru/PHP/images/ris74_1.jpg  
Рис.1. Общий вид функции **strtoupper()**

Строка, которую необходимо преобразовать, задается в качестве параметра.

Данные функции выполняют следующие операции (таблица 1).

|  |  |
| --- | --- |
| Таблица 1. **Функции преобразования строк** | |
| **Название функции** | **Операция, выполняемая этой функцией** |
| **strtolower()** | Переводит строку в нижний регистр |
| **strtoupper()** | Переводит строку в верхний регистр |
| **ucfirst()** | Переводит первый символ строки в верхний регистр |
| **ucwords()** | Переводит первый символ каждого слова строки в верхний регистр |

Приведем пример, характеризующий одновременно работу всех функций, содержащихся в таблице:

**<?php**

**$string = "мама купила красную шапку";**

**$string1 = "ШКОЛА ЗАКРЫВАЕТСЯ НА РЕМОНТ";**

**$strnew = strtolower($string1);**

**$strnew1 = strtoupper($string);**

**$strnew2 = ucfirst($string);**

**$strnew3 = ucwords($string);**

**echo 'Первоначальная строка $string: '.$string."<br>";**

**echo 'Первоначальная строка $string1: '.$string1."<br>". "<br>";**

**echo 'Преобразование строки $string1 с помощью функции strtolower():'."<br>";**

**echo $strnew."<br><br>";**

**echo 'Преобразование строки $string с помощью функции strtoupper():'."<br>";**

**echo $strnew1."<br><br>";**

**echo 'Преобразование строки $string с помощью функции ucfirst():'."<br>";**

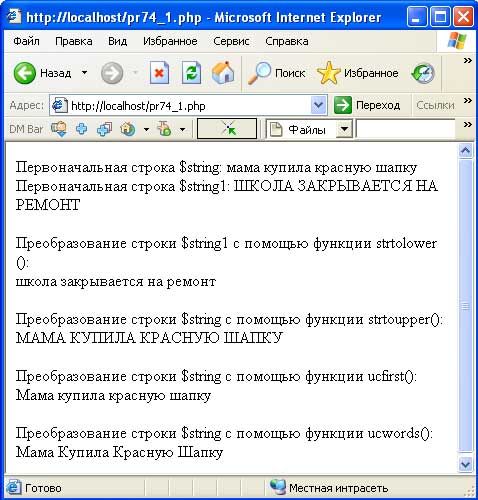
**echo $strnew2."<br><br>";**

**echo 'Преобразование строки $string с помощью функции ucwords():'."<br>";**

**echo $strnew3;**

**?>**

Результат выполнения этой программы смотрите на рисунке 1.

  
Рис.2. Пример применения функций **strtolower(), strtoupper(), ucfirst(), ucwords()**

Как видно в первом случае, функция перевела все символы строки в верхний регистр. Во втором случае все наоборот, все буквы строки переведены в нижний регистр. В третьем случае только первый символ строки переведен в верхний регистр. Последний случай показывает, что при помощи функции **ucwords()** первый символ каждого слова строки переведен в верхний регистр.

#### Операторы включения

##### include

Оператор *include* позволяет включать код, содержащийся в указанном файле, и выполнять его столько раз, сколько программа встречает этот *оператор*. Включение может производиться любым из перечисленных способов:

include 'имя\_файла';

include $file\_name;

include ("имя\_файла");

**Пример 3.9.** Пусть в файле c у нас хранится набор каких-то параметров и функций. Каждый раз, когда нам нужно будет использовать эти параметры (функции), мы будем вставлять в текст нашей основной программы команду include 'c'.

c

<?php

$user = "Вася";

$today = date("d.m.y");

/\* функция date() возвращает дату

и время (здесь – дату в формате

день.месяц.год) \*/

?>

include.php

<?php

include ("c");

/\* переменные $user и $today заданы в файле

c. Здесь мы тоже можем ими

пользоваться благодаря команде

include("c") \*/

echo "Привет, $user!<br>";

// выведет "Привет, Вася!"

echo "Сегодня $today";

// выведет, например, "Сегодня 7.07.05"

?>

**Пример 3.9. Использование оператора включения include (, )**

Заметим, что использование оператора *include* эквивалентно простой вставке содержательной части файла c в код программы include.php. Может быть, тогда можно было в c записать простой текст без всяких тегов, указывающих на то, что это php-код? Нельзя! Дело в том, что в момент вставки файла происходит переключение из режима обработки PHP в режим HTML. Поэтому код внутри включаемого файла, который нужно обработать как PHP-скрипт, должен быть заключен в соответствующие теги.

Поиск файла для вставки происходит по следующим правилам.

1. Сначала ведется поиск файла в *include\_path* относительно текущей рабочей директории.
2. Если файл не найден, то поиск производится в *include\_path* относительно директории текущего скрипта.
3. Параметр *include\_path*, определяемый в файле настроек PHP, задает имена директорий, в которых нужно искать включаемые файлы.

Например, ваш *include\_path* это . (то есть текущая директория), текущая рабочая директория это /www/. В основной файл include.php вы включаете файл my\_dir/a.php, который в свою очередь включает b.php. Тогда парсер первым делом ищет файл b.php в директории /www/, и если такового нет, то в директории /www/my\_dir/.

Если файл включен с помощью *include*, то содержащийся в нем код наследует область видимости переменных строки, где появился *include*. Любые переменные вызванного файла будут доступны в вызывающем файле с этой строки и далее. Соответственно, если *include* появляется внутри функции вызывающего файла, то код, содержащийся в вызываемом файле, будет вести себя так, как будто он был определен внутри функции. Таким образом, он унаследует область видимости этой функции. Хотя мы и не знакомились еще с понятием функции, все же приводим здесь эти сведения в расчете на интуитивное его понимание.

**Пример 3.10**. Пусть файл для вставки c останется таким же, а include.php будет следующим:

<?php

function Footer(){

// объявляем функцию с именем Footer

include ("c");

/\* включаем файл c.

Теперь его переменными можно пользоваться,

но только внутри функции \*/

$str = "Сегодня: $today <br>";

$str .= "<a

href='mailto:help@'>Страницу

создал $user</a>";

echo "$str";

}

Footer();

// вызываем функцию Footer(). Получим:

//Сегодня: 08.07.05

//Страницу создал Вася

echo "$user, $today";

// выведет запятую, так как

// эти переменные видны только

// внутри функции

?>

**Пример 3.10. Область видимости при использовани include (, )**

Кроме локальных файлов, с помощью *include* можно включать и внешние файлы, указывая их url-адреса. Данная возможность контролируется директивой url\_fopen\_wrappers в файле настроек PHP и по умолчанию, как правило, включена. Но в версиях PHP для Windows до PHP 4.3.0 эта возможность не поддерживается совсем, вне зависимости от url\_fopen\_wrappers.

include() – это специальная языковая конструкция, поэтому при использовании внутри условных блоков ее нужно заключать в фигурные скобки.

<?php

/\* Это неверная запись. Получим ошибку.

Мы же вставляем не одну команду,

а несколько, они только записаны

в другом файле \*/

if ($condition) include("first.php");

else include("second.php");

// А вот так правильно.

if ($condition){ include("first.php"); }

else { include("second.php"); }

?>

**Пример 3.11. Использование include() (, )**

При использовании *include* возможны два вида ошибок – ошибка вставки (например, нельзя найти указанный файл, неверно написана сама команда вставки и т.п.) или ошибка исполнения (если ошибка содержится во вставляемом файле). В любом случае при ошибке в команде *include* исполнение скрипта не завершается.

##### require

Этот оператор действует примерно так же, как и #include в C++. Все, что мы говорили о *include*, лишь за некоторыми исключениями, справедливо и для *require*. *Require* также позволяет включать в программу и исполнять какой-либо файл. Основное отличие *require* и *include* заключается в том, как они реагируют на возникновение ошибки. Как уже говорилось,*include* выдает предупреждение, и работа скрипта продолжается. Ошибка в *require* вызывает фатальную ошибку работы скрипта и прекращает его выполнение.

*Условные операторы* на require() не влияют. Хотя, если строка, в которой появляется этот оператор, не исполняется, то ни одна строка кода из вставляемого файла тоже не исполняется. Циклы также не влияют на require(). Хотя код, содержащийся во вставляемом файле, является объектом цикла, но вставка сама по себе происходит только однажды.

В реализациях PHP до версии 4.0.2 использование require() означало, что интерпретатор обязательно попытается прочесть вставляемый файл.

*require*, как и *include*, при использовании внутри условных блоков нужно заключать в фигурные скобки.

#### Решение задачи

И наконец, вернемся к задаче, сформулированной в начале лекции. Мы хотим создать программу, которую можно было бы использовать для отправки писем (или просто для их генерации) с приглашениями на различные мероприятия множеству пользователей. В предыдущей лекции уже рассматривался подобный случай. Сейчас мы вынесем всю информацию о людях и событиях в отдельный файл data.php и напишем программу, не зависящую (ну, может, совсем чуть-чуть зависящую) от этой информации и ее структуры. В этом случае для того, чтобы, например, расширить список адресатов, не нужно будет изменять скрипт, генерирующий приглашения. Кроме того, можно будет использовать информацию о людях и событиях в других скриптах. В самом скрипте, генерирующем приглашения letters.php, мы использовали *условные операторы*, циклы, *require* и другие изученные ранее конструкции.

<?php

define("SIGN","С уважением, Вася");

// пусть наша подпись

// будет константой

// информация о событиях

$events = array(

"f" => "день открытых дверей",

"o" => "открытие выставки",

"p" => "бал выпускников");

// имеющаяся информация о людях

// (имя и электронный адрес)

$people = array(

"ivan" => array(

"name" => "Иван Иванович",

"email"=>"user\_ivan@"),

"pit" => array(

"name" => "Петр Петрович",

"email" => "user\_petr@"),

"semen" => array(

"name" => "Семен Семенович"));

// кто куда приглашается

$who\_where["ivan"] = "o" ;

// Иван – на выставку

$who\_where["pit"] = "p";

// Петр – на бал

?>

**Листинг 3.12. data.php (, )**

<?php

require("data.php");

// включаем файл с данными о событиях

foreach($people as $key => $man\_info){

// для каждого человека делаем следующее:

$event\_key = $who\_where[$key];

// получаем событие,

// на которое он приглашается

if ($event\_key<>""){

foreach($man\_info as $key1 => $info){

// получаем имя и email

// конкретного человека

if ($key1=="name")

$str = "Уважаемый (ая), $info";

if ($key1=="email") $email = $info;

}

// составляем приглашение

$str .= "<br>Приглашаем Вас на ".

$events[$event\_key];

switch ($event\_key){

// в зависимости от события

// добавляем какую-нибудь строчку

case "f":

$str .= "<br>Подтвердите Ваше

участие по телефону!";

break;

case "o":

$str .= "<br>Приходите за 15

минут до открытия!";

break;

case "p":

$str .= "<br>Не забудьте подарок :-)";

break;

}

$str .= "<br>" . SIGN . "<hr>";

// добавляем подпись

echo $str; // вводим приглашение на экран

/\* если у вас настроена отправка почты с

помощью PHP, то письмо можно отправить

командой mail($email,"Letter",$str); \*/

}

}

?>

**Классы и объекты**

***Объект*** – это структурированная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии, *класс* – это описание таких *объектов* и действий, которые можно с ними выполнять.

В PHP *класс* определяется с помощью следующего синтаксиса:

class Имя\_класса{

var $имя\_свойства;

/\*список свойств\*/

function имя\_метода( ){

/\* определение метода \*/

}

/\*список методов\*/

}

Имена *свойств* *объектов* *класса* объявляются с помощью ключевого слова var, *методы*, применимые к *объектам* данного *класса*, описываются функциями. Внутри определения *класса* можно использовать ключевое слово *this* для обращения к текущему представителю *класса*.

Например, нам нужно создать *класс*, описывающий категорию статей. У каждой статьи имеются такие *свойства*, как название, автор и краткое содержание. Какие действия мы хотим совершать со статьями? Возможно, нам понадобится задавать значения перечисленным *свойствам* статьи, отображать статью в браузере. Тогда определение этого *класса* может выглядеть следующим образом:

<?

class Articles { // Создаем класс Статей

var $title;

var $author;

var $description;

// метод, который присваивает значения

// атрибутам класса

function make\_article($t, $a, $d){

$this->title = $t;

$this->author = $a;

$this->description = $d;

}

//метод для отображения экземпляров класса

function show\_article(){

$art = $this->title . "<br>" .

$this->description .

"<br>Автор: " . $this->author;

echo $art;

}

}

?>

Итак, для описания физических *объектов* типа "статья" мы создали *класс* с именем *Articles*, состоящий из трех переменных, содержащих характеристики статьи, и двух функций для создания конкретной статьи и для ее отображения.

Как известно, работая с PHP, можно периодически переключаться в режим HTML. В этом случае программа состоит из нескольких кусков (блоков) кода. Определение *класса* нельзя разносить по разным блокам php-кода и тем более по разным файлам. То есть если написать:

<?php

class Articles { // Начало описания класса

var $title;

?>

<?php

// продолжение описания класса

function show\_article(){

// содержание метода

}

} // конец описания класса

?>

то программа не будет работать корректно.

Несколько замечаний по поводу имен *классов*. Имя *класса* должно удовлетворять правилам именования *объектов* в языке PHP, но есть ряд имен, которые зарезервированы разработчиками для своих целей. В первую очередь это имена, начинающиеся с символа подчеркивания "\_". Для создания *классов* и функций нельзя использовать такие имена. Кроме того, зарезервировано имя stdClass, поскольку оно используется внутри движка PHP.

Часто некоторым атрибутам *класса* бывает необходимо присваивать значения сразу после создания представителя *класса*. Когда мы создавали*класс* статей, для присваивания значений атрибутам ( *свойствам* ) *класса* мы использовали специальную функцию make\_article(). Вообще говоря, мы поступили не совсем верно, потому что занялись изобретением велосипеда. Специально для задания начальных значений атрибутам*класса* существует два стандартных *метода*. В *PHP4* можно инициализировать значения с помощью оператора var или с помощью функции*конструктора*. С помощью var можно инициализировать только константные значения. Для задания не константных значений используют функцию*конструктор*, которая вызывается автоматически, когда *объект* конструируется из *класса*. Функция- *конструктор* должна иметь имя, совпадающее с именем всего *класса*, в котором она определена.

Приведем пример. Допустим, при создании *объекта* "статья" мы хотим установить его *свойства* следующим образом: автора – равным строке "Иванов", название и краткое содержание – соответствующим элементам глобального массива $*\_POST*, а дату публикации статьи – текущей дате. Тогда следующее описание *класса* не является корректным в *PHP4*:

<?

class Articles { // Создаем класс Статей

var $title= $\_POST["title"];

var $author = "Иванов";

var $description = $\_POST["description"];

var $published = date("Y-m-d");

// метод, который присваивает значения

// атрибутам класса

}

?>

А вот такое *описание класса в PHP4* будет работать так, как нужно:

<?

class Articles { // Создаем класс Статей

var $title;

var $author = "Иванов";

var $description;

var $published;

// метод, который присваивает значения

// атрибутам класса

function Articles(){

$this->title = $\_POST["title"];

$this->description = $\_POST["description"];

$this ->published = date("Y-m-d");

}

//метод для отображения экземпляров класса

function show\_article(){

$art = $this->title . "<br>" .

$this->description .

"<br>Автор: " . $this->author;

echo $art;

}

}

?>

Отметим, что в *PHP3* и *PHP4* *конструкторы* работают по-разному. В *PHP3* функция становилась *конструктором*, если она имела то же имя, что и *класс*, а в *PHP4* – если она имеет то же имя, что и *класс*, в котором она определена. Разница в подходах видна, когда один *класс* расширяет другой и происходит *наследование* *свойств* и *методов* *базового класса*. Но об этом мы поговорим чуть позже. В *PHP5* *конструктор* *класса* именуется\_\_construct. Кроме того, в *PHP5* появились и *деструкторы* – функции, которые вызываются автоматически перед уничтожением *объекта*. В *PHP5*функция-*деструктор* должна быть названа \_\_destruct.

**Объекты**

В одной из первых лекций мы упоминали о существовании в PHP такого типа данных, как *объект*. ***Класс*** – это описание данных одного типа, данных типа *объект*. *Классы* являются как бы шаблонами для реальных переменных. Переменная нужного типа создается из *класса* с помощью оператораnew. Создав *объект*, мы можем применять к нему все *методы* и получать все *свойства*, определенные в описании *класса* . Для этого используют такой синтаксис: $имя\_объекта->название\_свойства или $имя\_объекта->название\_метода(*список аргументов*). Заметим, что перед названием *свойства* или *метода* знак $ не ставят.

<?php

$art = new Articles;

// создаем объект $art

echo ($art ->title);

// выводим название объекта $art

$another\_art = new Articles;

// создаем объект $another\_art

$another\_art->show\_article();

// вызываем метод для

// отображения объекта в браузер

?>

Пример 6.1. Доступ к методам и свойствам объекта

Каждый из *объектов* *класса* имеет одни и те же *свойства* и *методы*. Так, у *объекта* $*art* и у *объекта* $another\_art есть *свойства* title,description, author и *методы* *Articles*(), show\_article(). Но это два разных *объекта*. Представим себе *объект* как директорию в файловой системе, а его характеристики – как файлы в этой директории. Очевидно, что в каждой директории могут лежать одинаковые файлы, но тем не менее они считаются различными, поскольку хранятся в разных директориях. Точно так же *свойства* и *методы* считаются различными, если они применяются к разным *объектам*. Чтобы получить нужный файл из директории верхнего уровня, мы пишем полный путь к этому файлу. При работе с *классами* нужно указывать полное имя функции, которую мы хотим вызвать. Директорией верхнего уровня в PHP будет пространство глобальных переменных, а путь указывается с помощью разделителя ->. Таким образом, имена $*art*->title и $another\_art->titleобозначают две разные переменные. Переменная в PHP имеет только один знак доллара перед именем, поэтому нельзя писать $*art*->$title. Эта конструкция будет рассмотрена не как обращение к *свойству* title *объекта* $*art*, а как обращение к *свойству*, имя которого задано переменной $title (например, $*art*->"" ).

<?php

$art->title = "Введение в Internet";

// так можно установить

// значение свойства объекта

$art->$title = "Введение в Internet";

// так нельзя установить

// значение свойства объекта

$property = "title";

$art->$property = "Введение в Internet";

// так можно установить значение

// свойства объекта

?>

Пример 6.2. Установка значений свойств

Создавая *класс*, мы не можем знать, какое имя будет иметь *объект* этого *класса*, тем более что *объектов* может быть много и все могут иметь разные имена. Соответственно мы не знаем, как обращаться к *объекту* внутри определения *класса*. Для того чтобы иметь доступ к функциям и переменным внутри определения *класса*, нужно использовать псевдопеременную $this. Например, $this->title возвращает значение*свойства* title у текущего *объекта* данного *класса*. Иногда эту переменную предлагают читать как "мое собственное" (к примеру, по отношению к *свойству* ).

**Наследование**

**extends**

Механизм *наследования* – очень важная часть всего *объектно-ориентированного подхода*. Попытаемся объяснить его суть на примере. Допустим, мы создаем описание человека. Очевидно, что сделать это мы можем по-разному, в зависимости от того, для чего нужно это описание. Можно описать человека как программиста: он знает такие-то языки программирования, операционные системы, участвовал в стольких-то проектах. Однако если человек программист, то он не перестает быть человеком вообще, т.е. он имеет имя, фамилию, место жительства и т.п. Если перевести наши рассуждения в термины объектно-ориентированного программирования, то можно сказать, что мы описали два *класса* – *класс*людей и *класс* программистов, каждый со своими *свойствами* и *методами*. Причем *класс* программистов, очевидно, обладает всеми *свойствами**класса* людей и при этом имеет свои специфические характеристики, т.е. *класс* программистов является подклассом *класса* людей. Так, если у человека вообще есть имя, то у программиста оно тоже должно быть, но не наоборот. Кроме программистов можно выделить еще множество*классов* по профессиональной принадлежности людей. И все они будут подклассами *класса* людей. Часто на практике удобно определять общий*класс*, который может использоваться сразу в нескольких проектах (например, *класс* людей или личностей), и адаптировать его для специфических нужд каждого проекта (например, как *класс* программистов). Как это можно реализовать? С помощью механизма расширений. Любой *класс* может быть расширением другого *класса*. Расширяющий (или производный) *класс*, кроме тех *свойств* и *методов*, которые описаны в его определении, имеет все функции и *свойства* основного ( *базового класса* ). В нашем примере *класс* программистов – расширяющий, а *класс* всех людей – базовый. Из *класса* нельзя удалить никакие существующие *свойства* и функции, *класс* можно только расширить. *Расширяющий класс* в *PHP4* всегда зависит только от одного *базового класса*, поскольку множественное *наследование* в PHP не поддерживается. Расширяются *классы* в PHP с помощью ключевого слова *extends*.

<?php

class Person { // определяем класс Личности

var $first\_name; // имя личности

var $last\_name; // фамилия личности

function make\_person($t,$a){

// метод устанавливает

// значения имени и фамилии объекта

$this->first\_name = $t;

$this->last\_name = $a;

}

function show\_person(){

// метод отображает информацию о личности

echo ("<h2>" . $this->first\_name . " " .

$this->last\_name . "</h2>");

}

}

class Programmer extends Person{

// определяем класс

// Programmer, расширяющий Person

var $langs = array ("Lisp");

// константным массивом

// задать переменную в var можно

function set\_lang($new\_lang){

// метод добавляет еще

// один язык к списку известных

$this->langs[] = $new\_lang;

}

}

?>

Пример 6.3. Использование механизма наследования

*Класс* Programmer имеет те же переменные и функции, что и *класс* Person, плюс переменную $langs, в которой содержится список изученных программистом языков, и функцию set\_lang для добавления еще одного языка к списку изученных. Создать представителя *класса*программистов можно обычным способом с помощью конструкции new. После этого можно устанавливать и получать список языков, которые знает программист, и в то же время можно использовать функции, заданные для *класса* Person, т.е. устанавливать и получать имя и фамилию программиста и отображать сведения о нем в браузере:

<?php

$progr = new Programmer;

$progr->set\_lang("PHP");

// методы, определенные для

// класса Programmer

print\_r ($progr->langs);

// методы, определенные для класса Person

$progr->make\_person("Bill","Gates");

$progr->show\_person();

?>

Отношения, в которых состоят созданные нами *классы* Person и Programmer, называют также *отношениями родитель–потомок*. *Класс*Person – родитель, а его потомки, такие как *класс* Programmer, создаются, основываясь на нем, с помощью расширений. Любой *класс* может стать родительским и соответственно породить потомков.

Порядок определения *классов* имеет значение. Нельзя сначала определить *класс* Programmer, *расширяющий класс* Person, а уже потом сам*класс* Person. *Класс* должен быть определен перед тем, как он будет использоваться (расширяться).

**Конструкторы**

Теперь, после знакомства с механизмом *наследования* в PHP, мы можем прокомментировать различие между *конструкторами* *PHP4* и *PHP3* и более подробно рассказать о *конструкторах* вообще. Напомним, что в *PHP3* ***конструктор*** – это функция, имя которой совпадает с именем *класса*. А в*PHP4* – функция, имя которой совпадает с именем *класса*, в котором она определена.

<?php

class Programmer extends Person{

// определяем класс

// Programmer, расширяющий Person

var $langs = array ("Lisp");

function Programmer(){

// этот конструктор будет

// работать и в PHP3, и в PHP4

$this->make\_person("Иван","Петров");

}

}

?>

Пример 6.4. Использование конструктора

Здесь функция Programmer() является *конструктором*, т.е. выполняется сразу после создания любого представителя *класса* Programmer, задавая ему имя "Иван" и фамилию "Петров". *Конструкторы*, как и любые другие функции, могут иметь аргументы. В этом случае, создавая представителя *класса*, нужно указать значения этих параметров. Аргументы *конструктора* могут иметь и значения по умолчанию. Если все аргументы имеют значения по умолчанию, тогда можно создавать экземпляр *класса* без параметров.

<?php

class Programmer extends Person{

// определяем класс

// Programmer, расширяющий Person

var $langs = array ("Lisp");

function Programmer($n = "Иван",

$f = "Петров"){

// это конструктор

$this->make\_person($n,$f);

}

}

$default\_progr = new Programmer();

// создаст программиста Ивана Петрова

$new\_progr = new Programmer("Вася",

"Сидоров");

// создаст программиста Васю Сидорова

print\_r($new\_progr);

/\* выведет информацию о переменной

$new\_progr, т.е. свойства объекта

и их значения \*/

?>

Пример 6.5. Использование конструктора

Приведенные примеры будут работать и в *PHP3*, и в *PHP4*, конечно если дописать в них определение *базового класса* Person. Допустим, ситуация немного другая: *конструктор* имеется только у *базового класса* Person:

<?php

class Person { // определяем класс Личности

var $first\_name;

var $last\_name;

function Person($t,$a){ // конструктор

$this->first\_name = $t;

$this->last\_name = $a;

}

/\* ... \*/

}

class Programmer extends Person{

// определяем класс

// Programmer, расширяющий Person

var $langs = array ("Lisp");

function set\_lang($new\_lang){

$this->langs[] = $new\_lang;

}

}

$new\_progr = new Programmer("Вася",

"Сидоров");

?>

Что произойдет в этом случае при создании *объекта* *класса* Programmer, будет ли автоматически вызвана какая-либо функция? В *PHP3* ничего не произойдет, поскольку в этом *классе* нет функции с именем Programmer() (здесь *конструктор* – это функция, имя которой совпадает с именем*класса* ). В *PHP4* будет вызван *конструктор* *базового класса*, если он существует, т.е. вызовется функция Person() из *класса* Person (здесь*конструктор* – функция, имя которой совпадает с именем *класса*, в котором она определена).

Еще одна ситуация – в *базовом классе* есть функция, имя которой совпадает с именем *расширяющего класса*, а в *расширяющем классе* нет*конструктора*.

<?php

class Person { // определяем класс Личности

var $first\_name;

var $last\_name;

function Person($t,$a){ // конструктор

$this->first\_name = $t;

$this->last\_name = $a;

}

function Programmer($new\_lang){

echo "Я – программист";

}

}

class Programmer extends Person{

// определяем класс

// Programmer, расширяющий Person

var $langs = array ("Lisp");

function set\_lang($new\_lang){

$this->langs[] = $new\_lang;

}

}

$new\_progr = new Programmer("Вася",

"Сидоров");

?>

В этом случае *PHP3* вызовет в качестве *конструктора* функцию Programmer() из описания *класса* Person. Поскольку *конструктор* – это функция, у которой то же имя, что и у *класса*. И неважно, определена ли эта функция в самом *классе* или она наследуется из *базового класса*. В*PHP4* *класс* Programmer не будет иметь своего *конструктора*, поэтому вызовется *конструктор* *базового класса*.

Ни в PHP 3, ни в PHP 4 *конструктор* *базового класса* не вызывается автоматически из *конструктора* порожденного *класса*.

**Оператор ::**

Иногда внутри описания *класса* возникает необходимость сослаться на функции или переменные из *базового класса*. Бывает, что нужно ссылаться на функции в *классе*, ни один представитель которого еще не создан. Как быть в таком случае? В *PHP4* для этого существует специальный оператор"::"

Например, вот так можно вызвать в описании *класса* Programmer функцию show\_name() из *базового класса* Person и функциюsay\_hello(), заданную в описании *класса* Programmer, когда ни один *объект* этого *класса* еще не был создан:

<?php

class Person { // определяем класс Личности

var $first\_name;

var $last\_name;

function Person($t,$a){ // конструктор

$this->first\_name = $t;

$this->last\_name = $a;

}

function show\_name(){

// метод отображает информацию о личности

echo ("Меня зовут, " .

$this->first\_name . " " .

$this->last\_name . "!<br>");

}

}

class Programmer extends Person{

// определяем класс

// Programmer, расширяющий Person

function set\_lang($new\_lang){

// метод добавляет еще

// один язык к списку известных

$this->langs[] = $new\_lang;

Person::show\_name();

// вызываем функцию из базового класса

echo "И я знаю теперь еще и " .

$new\_lang;

}

function show\_name(){

echo ("Я программист, " .

$this->first\_name . " " .

$this->last\_name . "!<br>");

}

function say\_hello(){

echo "Привет!<br>";

}

}

Programmer::say\_hello();

// вызываем функцию, когда ни

// один объект ее класса еще не создан

$new\_progr = new Programmer("Вася","Сидоров");

$new\_progr->set\_lang("PHP");

?>

В результате работы этой программы получим следующее:

Привет!

Меня зовут Вася Сидоров!

И я знаю теперь еще и PHP

С помощью команды Programmer::say\_hello(); мы вызываем функцию say\_hello *класса* Programmer как таковую, а не как *метод*, применяемый к *объекту* данного *класса*. В этот момент переменных *класса* нет. Поэтому функции, вызываемые до создания *объекта*, не могут пользоваться переменными *класса* и конструкцией *this*, но могут пользоваться локальными и глобальными переменными.

В определении *класса* Programmer мы переопределили функцию show\_name(), поэтому вызвать функцию show\_name() из *базового класса*Person можно только с помощью оператора "::" Вообще говоря, внутри определения *класса* мы можем вызывать любые *методы* и *свойства*, заданные в его *базовом классе* с помощью обычного $this, если только порожденный *класс* не переопределяет эти *свойства* и *методы*, как в нашем примере.

**Оператор parent**

В приведенном выше примере, обращаясь к *базовому классу*, мы использовали его имя (мы писали Person::show\_name() ). Это не совсем удобно, потому что имя *класса* или иерархия *классов* может измениться, и тогда придется переписывать код описаний всех *классов* с тем, чтобы привести используемые в них имена в соответствие с новой иерархией. Чтобы избежать подобной ситуации, вместо имени *базового класса* нужно использовать ключевое слово parent (например, parent::show\_name() ). Parent ссылается на *класс*, прописанный после *extends* в объявлении вашего *класса*. Поэтому если вдруг иерархия *классов* изменится, то достаточно будет внести изменения в имена, указанные после*extends* в описаниях *классов*.